AD-A120 319 ADVANCED INFORMATION AND DECISION SYSTEMS MOUNTAIN V--ETC F/G 9/2
DESIGN OF AN INTELLIGENT PROGRAM EDITOR.(U)
SEP 82 D G SHAPIRO, B P MCCUNE, G A WILSON N00014-82-C-0119
UNCLASSIFIED AI/DS-TR-3023-1 NL

1 OF 2
AD A
120319

12

AI&DS

AD A120319

TR 3023-1

DESIGN OF AN INTELLIGENT PROGRAM EDITOR

Daniel G. Shapiro, Brian P. McCune, Gerald A. Wilson

September 1982

Final Technical Report for 1 January - 31 July 1982

Prepared for

Office of Naval Research
800 N. Quincy Street
Arlington, VA 22217

DCASMA/SF
1250 Bayhill Drive
San Bruno, CA 94066

ADVANCED INFORMATION & DECISION SYSTEMS
Mountain View, CA 94040

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR 3023-1	2. GOVT ACCESSION NO. AD-A120 319	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Design of an Intelligent Program Editor		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report 1 January - 31 July 1982
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Daniel G. Shapiro, Brian P. McCune, Gerald A. Wilson		8. CONTRACT OR GRANT NUMBER(s) N00014-82-C-0119
9. PERFORMING ORGANIZATION NAME AND ADDRESS Advanced Information & Decision Systems 201 San Antonio Circle, Suite 286 Mountain View, CA 94040		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research 800 N. Quincy St. Arlington, VA 22217		12. REPORT DATE September 1982
		13. NUMBER OF PAGES 113
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) DCASMA San Francisco 1250 Bayhill Drive San Bruno, CA 94066		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

program editing, ADA, ADA Programming Support Environment (APSE), artificial intelligence (AI), knowledge-based system, intelligent user interface, program reference language, program searching, program manipulation, semantic program analysis, documentation assistant, style analysis.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report discusses results of a project to develop a functional design for and assess the feasibility of an intelligent program editor for ADA and other programming languages. The editor will support program development and maintenance activities by providing advanced techniques for searching through programs, manipulating programs, analyzing programs for potential errors and good style, and maintaining structured documentation. These techniques are based on knowledge-based systems technology from the field of artificial intelligence.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT Feasibility of the program editor is demonstrated by a functional design and an initial implementation of the multiple knowledge bases representing a small program and a search (query) mechanism that uses them. The use of such an editor implies significant benefits for programmer productivity, program reliability, and life-cycle costs.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

List of Figures

1. Introduction

The development and maintenance of large software systems is both costly and time consuming. Typically large quantities of code are generated by a number of individuals. This code must be coordinated, interfaced, and made to work as a logical unit in order to achieve the intended effect. Because of the magnitude of this task, tools which can assist the developers with the creation, maintenance, documentation, and debugging of large software systems are quite important. In this project we have developed some tools which we believe are useful in the module-level development of large software systems.

It is our observation that the tools which currently exist do not adequately fill the needs of software developers. As is discussed further below, currently available tools tend to fragment the necessary development tasks into a collection of loosely coupled tasks, with little aid in either the structuring of these tasks or the utilization of shared information (such as the specifications for code modules used by many programmers). This results in higher software development costs, incomplete sharing of information among the developers, incomplete corporate memory, and greater than desired burdens upon the developers.

Domain knowledge and reasoning can be quite useful in a programming support environment. This is nicely illustrated with an example taken from an analogous situation. Suppose we had a technical manuscript which was to be typed. If the manuscript was given to a typist who spoke no English, the result might be, at best, a word-for-word typewritten copy of the manuscript. If the manuscript was given to an English-speaking typist, simple errors, such as misspellings and punctuation problems, might be fixed as a part of the typing process. If we gave our manuscript to an English teacher moonlighting as a typist, the result might well be a typed manuscript in which the prose had been smoothed and otherwise improved. If we were lucky enough to find a typist familiar with the domain of discourse of the manuscript, the result would likely have factual errors corrected and incomplete thoughts identified.

The problem of getting the manuscript typed with the best possible result is similar to the problem of writing a good program. You might start by selecting the type of editor to use for entering the program text. A standard text editor would be comparable to the non-English-speaking typist; text would appear exactly as it was typed, with no enhancements. The English-speaking typist could be compared to a syntax-oriented editor, one which can eliminate syntactic program errors and misspelled words. The remaining two typists have a significant degree of knowledge about your task and understand how to apply that knowledge. The English teacher/typist knows about the language itself, but not about the content of the thoughts. This situation is comparable to a programming language-specific editor, one which applies knowledge about the domain of programming; the editor can help with general programming techniques, can catch certain types of semantic errors, can make style suggestions, and can improve the general flow of the program. The technical typist who understands the content of what is being said is analogous to an editor which utilizes knowledge about the application domain; it can help with domain specific techniques, such as algorithm development, and can catch certain types of pragmatic error which are dependent upon the specific application domain.

One possible approach to improve this situation is to provide the software developers with more powerful and more intelligent editing environments. By this we mean editors which have built-in knowledge about the nature of software development, the types of information which will be useful to the developers, ways in which information can be used, and the types of corporate memory required. The objective of this project was to investigate the feasibility of developing an intelligent program editor (IPE), and the types of assistance which might be provided by such a tool to software developers. By demonstrating the feasibility of developing an intelligent editor, the way would be cleared for more extensive pursuit of sophisticated editing tools that could improve the productivity of software developers, improve the reliability of a system throughout its life, and reduce the costs of software development and maintenance.

1.1. Approach

This feasibility study was performed in four stages:

1. Identify functional properties -- By a careful examination of the tools now available to support software development, and the activities required to accomplish the development of large software systems, the functional properties which would be of value were developed. Using a number of examples of the uses of an intelligent editor, these functional suggested properties were further scrutinized and refined. The functional properties were then examined in more detail to determine the types of information which would be required, potential information

representations, and anticipated technologies that might be required.

2. Examine relevant technologies -- Using this high level functional specification, each of the potentially relevant technologies was examined to determine what functions might be supported with existing or small extensions of existing technology. At the same time those functional properties which will require significant technological advances were also identified. This provided the basis for a determination of both the long and short term feasibility of the development of an intelligent editing environment.

3. Develop functional design -- Having identified both the desired functional properties and the relevant technologies, a high level functional design of an intelligent editor was developed. This functional design represented the initial step in demonstrating the feasibility of developing an intelligent program editor.

4. Perform feasibility experiment -- As the final step in the determination of the feasibility, a primitive experimental editor was developed. This tool was used to demonstrate that, for a single small program, techniques could be developed to support a number of searching capabilities based upon the use of high level knowledge. Thus this searching capability demonstrated that at least some aspects of the envisioned intelligent program editor can be achieved.

1.2. Conclusions

A number of important results were achieved in this project:

1. A high level design for a powerful intelligent editor was developed. This editor provides a comprehensive software development environment and offers capabilities far beyond those of conventional editors.

2. The feasibility of developing such an editor was demonstrated both by the specification of the high level design and the implementation of a primitive portion of the editor capable c performing limited types of semantically based text searches.

3. The functional properties suggested for the intelligent edit will reduce the burden on programmers by providing contextually appropriate support for all aspects of the development of software systems. This is achieved by integrating all aspects of the software module development and maintenance process, thus allowing information to be provided once by the programmer, rather than separately for program specification, documentation, functional design, etc.

4. Support of the "corporate memory", i.e. the transfer of information, including both original documentation and additional annotations, from old to new people performing the same job, is facilitated directly by the editor through the incorporation of tools to guide programmers in the inspection and testing of existing code.

5. Greater compliance with desired software development guidelines can be achieved through the automatic application of tools to measure code style standards, documentation standards, algorithm choices, and design structures.

Thus the intelligent editor could provide improved capabilities for the software developers and at the same time reduce the life-cycle cost of software systems.

2. Design for an Intelligent Editor

The editing process can be considered from two different levels. First is the text symbol operation level, one at which we are concerned only with the physical manipulation of text strings. This is the only level which most traditional editors (line or screen) address. The second, and more important level is one in which the intent of the programmer is reflected in the nature of the text operations to be performed. This is the level which is alluded to by editors which permit "structures" of a program to be edited (for example Lisp s-expressions). While this does reflect the desire of the programmer to edit a program in a language whose syntax is understood, at least in part, by the editor, it does not capture any of the other very important aspects of the user's intent. For example, such an editor still cannot locate the portion of the program containing "the FOR loop using I as the controlling variable."

However, the motivation behind the creation of "structure" editors is important to understand, because it indicates many of the failings of conventional editors. People developed editor which could recognize some of the structure of a program in order to enable programmers to edit program text in semantically meaningful units. Thus movement and editing within the program text might be accomplished at the "program statement" level, rather than on the basis of lines of text on a screen. Thus the intent was to improve the utility of the editor by making it "smarter" than simple string editors.

While this approach provides some aid in the editing of single program modules, it does not aid in the editing of large collections of program modules which are inter-related. The reason is that knowledge about the syntax of the programming language is not sufficient to capture the information needed to support other types of editing operations. For example, syntax alone will not enable the editor to check that the call to a subroutine matches the syntax of the subroutine declaration. To accomplish such more semantically oriented tasks the editing system must have knowledge about both the syntax of the language and the way in which large software systems are constructed out of independently defined but inter-related modules.

Continuing along this line of thought quickly leads one to the conclusion that editors will be better when they incorporate sufficient understanding of the programmers' intent that they can support actions which more closely match the actions intended by the programmer. What we are searching for is not just collections of macro instructions which combine a number of simpler syntactic operations, but these together with actions determined by inference based upon knowledge about the programming environment. Consider again the example of subroutine usage checking of the previous paragraph.

For the editor to determine that it must check the correspondence between a subroutine declaration and an invocation of that subroutine, several steps must be achieved. First, it must recognize the syntax of the programming language sufficiently that it can identify both the subroutine declaration and an invocation of that subroutine. Second, it must be able to draw the inference that a reference to the subroutine in a "calling" statement is intended to cause the parameters in the "calling" statement to be inserted in place of the formal parameters of the "called" subroutine so that the subroutine may be executed. Third, the editor must infer that this application of the subroutine will only work properly if the parameters passed from the calling statement are consistent in number and type with the declared formal parameters of the subroutine. Fourth, the editor must "know" that it is unlikely that a programmer would deliberately cause a mismatch between the calling statement and the subroutine declaration, and thus a mismatch is most likely an error which should be reported to the programmer.

Traditionally the burden of such checking has fallen on compilers, linkers, or the programmer. But this is like building a bridge from both sides towards the middle without ever checking to assure that the ends will meet until you actually reach the middle. It is always easier and less time consuming for a programmer to correct an error as soon as it can be recognized, rather than waiting until sometime later. Finding and correcting the error at a later time always requires the programmer to re-establish the context of the code in error, so that he may infer the original intent.

The analogy from this example can be further extended by looking at the fact that the programmer is writing code in order to cause one or more tasks to be performed by the computer. The code which includes the subroutine call must be consistent with the purpose behind the code. Thus tools which would at least associate some description of the purpose behind the code with the code, and possibly even check for consistency between the purpose and the code, would be a great value to the programmer.

What we have tried to accomplish in this project is to consider the nature of software development in its fullest context. It is our belief that a major improvement in software development tools must be based upon the creation of editors which provide a full "environment" for development, rather than just tools for syntactic manipulation of text. The remainder of this chapter is a description of the characteristics of such an environment together with a design for an intelligent editor which would realize the desired environment.

An overview of the Intelligent Program Editor concept is shown in Figure 2.1. All of the interaction between the system and the user is handled by the user interface. All of the supporting tools exercised by the user are managed by the editing executive. Knowledge about the types of editing contexts in which a programmer might be operating, and the types of tools and information required to support that context, is provided in the programming context model accessed by the editing executive. The description of the actual programs together with all of the associated information (documentation, algorithm descriptions, functional descriptions, etc.) is maintained by the extended program model. Each of these major components is described in detail in the sections below.

2.1. A Context for Software Development (The Programming Context Model)

Consider editing in the broad perspective of all phases of the development of large software systems. Such editing includes the creation and modification of program text, documentation, and test data. While not all of this information is currently handled by computer based editors, it is all closely tied to the activities of software develop-

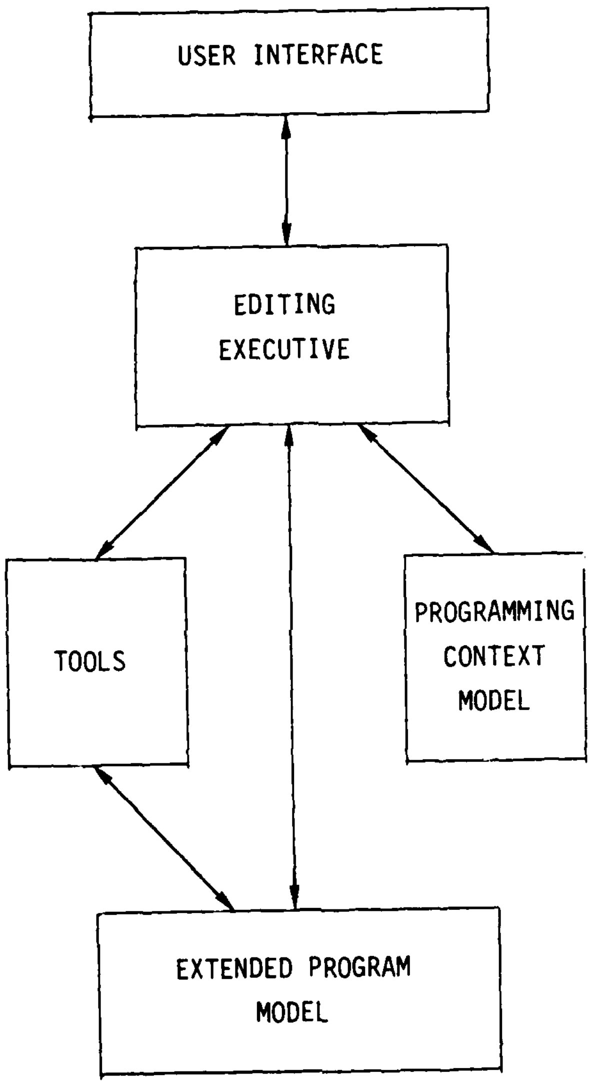


Figure 2.1: The Intelligent Program Editor (IPE)

ment, and it is all highly interwoven. Thus it would be of benefit to consider all of this information as a part of the complete description of a software system.

Taken from this perspective, one can describe the major activities comprising the editing of software systems in the terms shown in Figure 2.2. At the top level we have divided the activities of software development into four contexts: creation, debugging, modification, and exploration. These are all inter-related in the sense that they share component activities, but each represents a distinct intent associated with the user of the editor.

The general model of software system development used here is that large software systems are created from collections of distinct modules. These modules are brought together into a hierarchically organized tree or network which collectively performs the desired functions. Because there are a large number of modules, and some of the modules may be brought from standard libraries or other independent software systems, the complete system tends to be developed by many people, each with responsibilities for a logical segment of the complete system. This model of software development incorporates all aspects from the initial high level functional specification down through the detailed code generation and testing. In general it is assumed that a structured approach is being used, so that the development effort tends to be done as a recursive application of the stages shown in Figure 2.2. (In actual current development efforts this is not always true, especially in software system maintenance. However, it should be true, and tools should promote and facilitate this approach.)

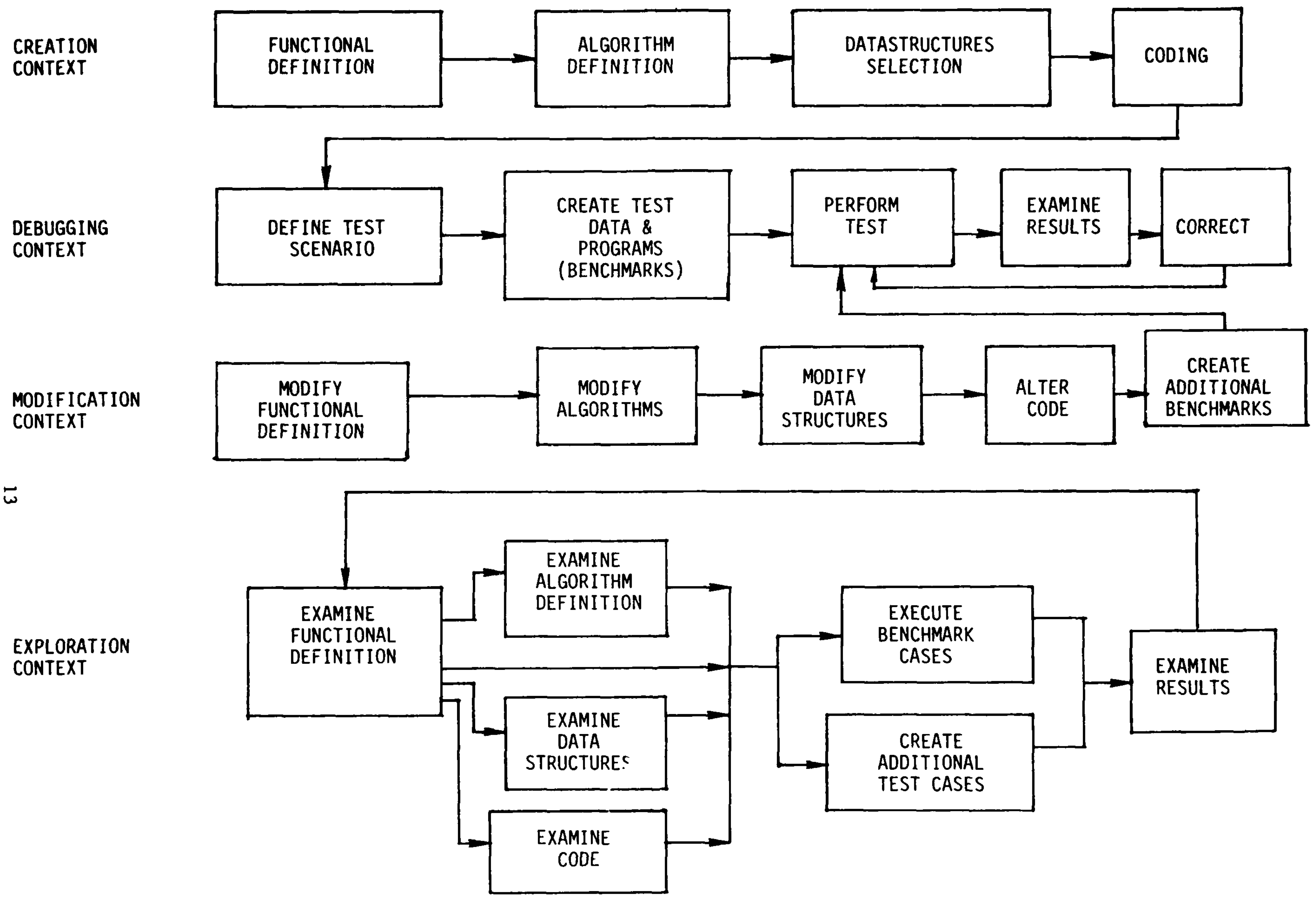


Figure 2.2 Major Editing Activities within Contexts

2.1.1. Software Creation

The first stage in the development of software is the definition of the functional properties which are to be exhibited by the end product. Typically these top level functions are then subdivided into supporting collections of functions, and so on until the desired degree of modularity has been achieved. This hierarchical collection of functions thus provides the specifications needed to permit the definition of algorithms which will achieve these functions, together with the data structures needed by those algorithms. Finally, the text comprising the actual code can be composed. Unlike conventional languages, the ADA language is designed to support and represent both functional design and code.

Notice that these stages have moved quite naturally from high level specification, and thus documentation, of the software through to the most detailed specification, the code. As is described further below, the editor can provide tools which not only support each of these stages of the software creation, but utilize information from one stage to assist the user in the next stage.

2.1.2. Software Debugging and Testing

Once the code has been created it must be tested to determine if it performs as intended, and repaired as necessary. To accomplish this appropriate collections of test data must be generated, and the results to be expected when the software system is exercised with this test data must also be determined and associated with the test data. As these represent standard cases, they should be preserved as "bench-marks" which may be used in both the initial debugging of the code and in testing of the code for various purposes t a latter time.

An important aspect of the debugging process is the accumulation of bench-mark cases. These test sets represent a part of the documentation of the software system, as well as tools for the testing and debugging of the software system being developed. By incorporating the development and use of the bench-marks directly within the context of the editor, both purposes can be properly supported.

2.1.3. Software Maintenance and Modification

Large software systems normally have a relatively long lifetime (e.g., averaging approximately 10 years in U.S. Air Force C3I software [Dean & McCune-82]). In general this is due to the fact that the cost of developing large systems dictates that the choice of problems to be addressed be limited to those for which the system will continue to serve a useful function for an extended period of time. However, some changes in the functions of system are almost inevitable, as the needs of the organization being supported change.

Changing an existing, running system requires a redesign from the functional definition on down (and often from the requirements level on down). This is one of the situations in which complete and accurate documentation of the system is crucial. An intelligent editing environment in which all levels of documentation are directly available on-line and easily accessed is thus an important tool in minimizing the work required to understand how the desired changes may be achieved, documenting the changes, altering the system code, and debugging the altered system.

2.1.4. Software Comprehension

The longer the time that a large software system is in use after the initial development of the system, the more likely it is that new programmers are involved with maintaining the system and that the original programmers no longer remember the details of the software system. The results in a need to train, or retrain, people whenever corrections or modifications to the system must be made. This "software comprehension" problem in fact appears to be the key problem in software maintenance today [Dean & McCune 82]. Such training includes examination of the documentation, exercising of example cases, exercising of new test cases, and detailed examination of the results of the execution of the various functions composing the system capabilities.

Here again it is clear that a fully integrated environment is needed, one in which all levels of documentation of the system, plus all of the code, is available to the programmers. This type of support is one which is usually overlooked in the tools provided to system developers, and yet it is this very re-training exercise which constitutes a major component of the expense of maintaining larger software systems over long periods of time.

2.2. Intelligent Program Editor Tools

The software development contexts described in the previous sections are the framework within which a user of the intelligent program editor operates. These contexts provide an environment which is both appropriate to the user's tasks and helpful in terms of guiding the user through the tasks needed. The guidance is provided by means of high level knowledge associated with each context. This knowledge describes the collection of steps which a user is expected to execute (though not necessarily in the order in which a particular user will accomplish those steps) by the time the job has been completed. Associated with each step is advice about the editor tools which might be invoked, the umstances under which they should be invoked, and who controls their ..vocation.

For example, as shown in Figure 2.3, as a user moves through the various steps cf program creation, the system will employ templates to aid in the functional definition steps, provide access to libraries of typical programming procedures to aid in the development of the detailed algorithms, provide general knowledge about the types of data structures which may be employed and some guidelines for the benefits and drawbacks of the various data structures, provide some automated checking for coding errors, and even partially automate the construction of code once the functional and algorithmic descriptions have been provided.

In the sections below we describe the major types of tools which we believe should ultimately be included in the intelligent program editor environment. Later in this report some ways in which some of these tools might be implemented and supported within the IPE environment are discussed. Our intent is not to imply that the tools of the IPE should be limited to those discussed here, but rather to suggest a broad variety of tools believed to be useful. As other appropriate tools become available they may also be included in the IPE framework.

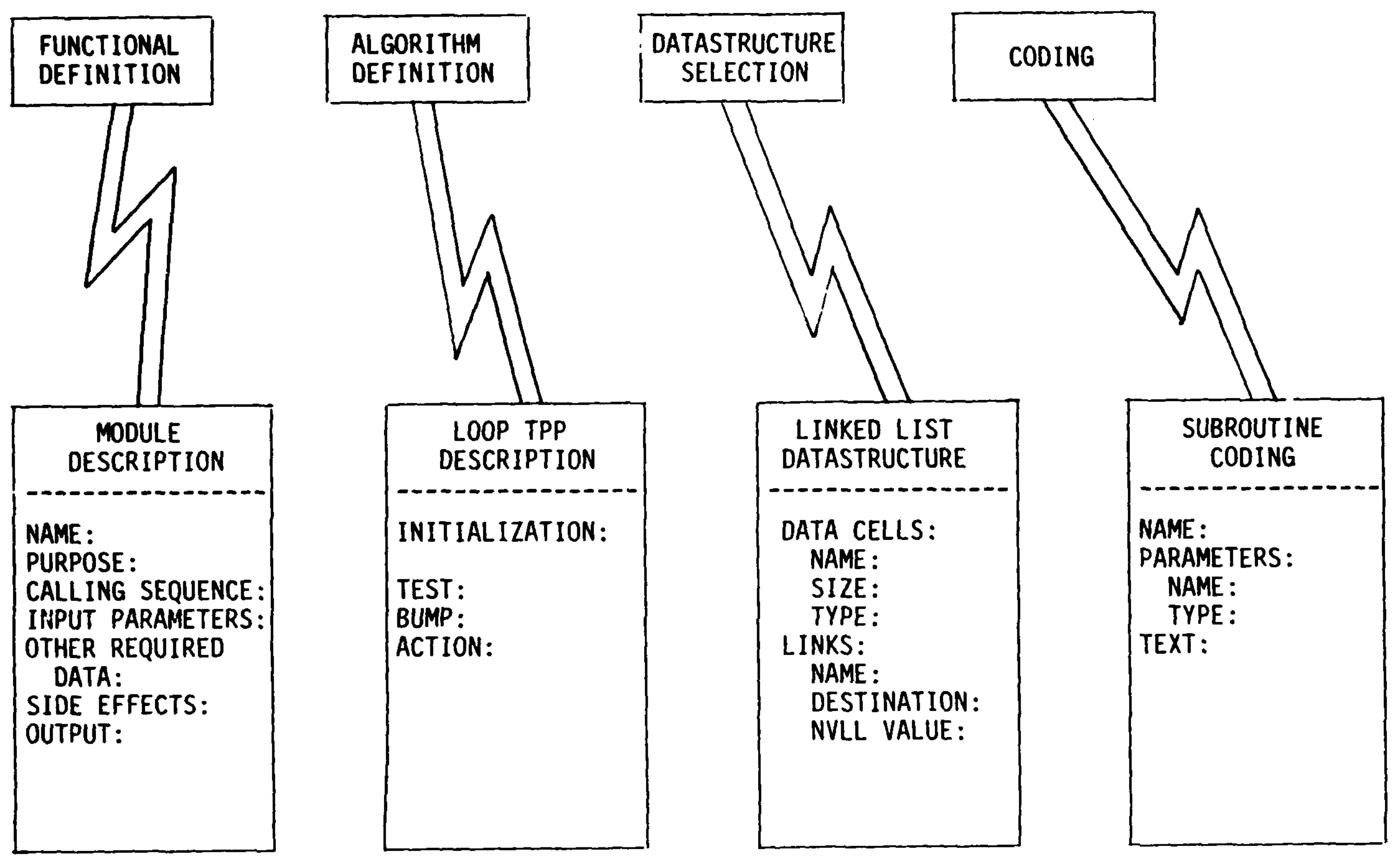


Figure 2.3 Some Templates for Program Creation Steps

2.2.1. Program Searching and Editing

Regardless of the other functions performed by the intelligent program editor, its principle purpose is to enable the creation and editing of text associated with all aspects of the definition of software. Therefore all of the capabilities for text manipulation available in advanced screen editors now used within many popular operating systems are included in the text editing facilities of the IPE.

A major failing of traditional editors is that the only searching capabilities provided are based upon text strings. Thus a user attempting to locate something like the "initialization of the variable TEST-SCORE within the FOR loop computing the sum of the test scores" can only do so by searching for text strings containing ´FOR´ or ´TEST-SCORE.´ Furthermore, if the user was searching for a description of the calling sequence and functions performed by a given subroutine, the only support provided by the editor would be to display the code, or any on-line documentation, when given the explicit names of the text files. Standard editors have no vocabulary to allow a user to express queries which utilize the logical structures of the program, rather than just the text structures. Furthermore, even if such queries could be expressed, the editors have no knowledge which would permit them to locate the appropriate files directly, to distinguish between on-line documentation and program text, or to determine the most appropriate source of information to satisfy the user´s search request.

Because the IPE employs an extended program definition (as explained further below) which includes all of the documentation describing the program, plus a number of special descriptions, as well as the code, a very broad variety of searches can be supported. Thus the IPE would be able to perform searches such as that of the example in the previous paragraph. Such searches may require combinations of textual, syntactic, semantic, application domain, and contextual cues to be fully executed. The IPE also will be able to determine appropriate types of information to return when choices are available. For example, given a request about a specific subroutine, such as that mentioned above, the IPE would be able to present either the source code or the high level documentation. The choice of which information to present would be made using general heuristics about the user's intent as reflected in the general context of operation (the programming context model), if no further instructions were provided by the user. An extensive discussion about these search capabilities and the means to achieve them is presented in the sections below.

2.2.2. Advanced Program Manipulation

Many editors provide collections of macro operations which can be used to perform operations which require multiple steps. For example, groups of lines of text can be deleted or moved from one position to another, global substitutions may be performed, etc. All of these operations are defined in terms of collections of string manipulations, and the editor operates without any knowledge of the implications of the actions or the user's intent.

With an intelligent editor that has a substantial amount of knowledge about both the semantic structure of programs and the semantics of meaningful operations, much better support can be provided to the user. For example it would be possible to provide operations that

```
I = 1
WHILE I ≤ N DO
   BEGIN
     .
     .
     .
   I = I + 2
   END
```

Figure 2.4 (a) The WHILE Loop Before Transformation

```
FOR I = 1 STEP 2 UNTIL N DO
   BEGIN
     .
     .
     .
     END
```

Figure 2.4 (b) The FOR Loop After Transformation would directly transform the "WHILE" loop shown in Figure 2.4(a) into the "FOR" loop shown in the second portion of that figure. Another type of operation might be one which interactively constructed a subroutine call by first requesting the name of the subroutine and then either prompting the user for each argument or providing a template to fill. The knowledge for this interactive process would be obtained automatically by the editor using the functional definition of the subroutine as previously provided to the system.

As with the simple syntactic manipulations provided with traditional editors, these semantically based manipulations will allow the user to accomplish complex, but well defined, tasks with a minimum of effort. These tools can also help to reduce errors in the code by allowing the documentation describing the code to be used directly in the course of the code generation.

There are certainly manipulations which might be desired that will be beyond the level of current technology. For instance, while it might be quite useful to have automatic translation of Lisp programs in Fortran, such transformations could not now be accomplished without substantial assistance from human users. However, as technology advances, these new and more powerful tools can be added to the IPE framework.

2.2.3. Documentation Assistance

References have been made many times in the text above to the assistance provided the user by the IPE in the generation and use of high level documentation of programs. This is as extremely important aspect of the IPE. The intent is to have tools which both aid the user in the construction of the documentation and avoid making the user provide what is semantically the same information but in two or more different forms. Note that our definition of documentation is quite broad. It includes everything from the high level specifications of the intended system functions to the comments associated with individual code modules. The documentation assistance tools will help in the process of writing documentation, using documentation, and keeping the documentation updated. These tools will provide support for incremental documentation (due to the incremental nature of software development and maintenance, the only way to keep documentation consistent with the code is to update the documentation incrementally as the code is changed.)

Both of the types of documentation found in the programming process will be handled: program comments and documents. Documents include a broad base of "non-program" text, including high-level design specifications, user manuals, test data, etc. Both types of documentation will be treated uniformly by the IPE, which is somewhat different than the standard practice of treating program comments as "lower class" documentation. Each section of documentation will able treated as a unique object. A program comment, previously considered to be a piece of text that happened to lie adjacent to some piece of code in a source file, will now be formally considered an object in its own right. All documentation will be associated (i.e. linked with) the code as a part of the knowledge structures of the extended program model. Documentation objects are structured (like templates or frames). Each object will have a functional property associated with it which specifies the relationship of the documentation to the code. This link will also have chronological information, author information, and other such historical data which will enable the proper historical records to be maintained.

```
                    FUNCTIONAL DESCRIPTION

  NAME:  MEAN

  PURPOSE:  COMPUTE OF TEST SCORES FOR ALL STUDENTS
            OF A GROUP

  CALLING SEQUENCE:  MEAN (SCORES, GROUP)

  INPUT PARAMETERS:  SCORES = UNORDERED LIST OF PAIRS,
                              EACH CONSISTING OF A
                              STUDENT_ID AND THE
                              CORRESPONDING SCORE
                     GROUP =  IDENTIFIER OF STUDENT GROUP
                              TO BE USED

  OTHER REQUIRED DATA:  (STUDENT_ID, GROUP) PAIRINGS

  SIDE EFFECTS:  NONE

  OUTPUT:  MEAN_TEST_SCORE AS REAL NUMBER
```

Figure 2.5 An Example Functional Definition Template

To understand some of the types of tools which can be provided, consider the example shown in Figure 2.5. Here the user has informed the system that he is creating new code modules. The system responds by providing the user with a template (shown as the underlined text) which suggests the type of information which should be included in the the functional description of the code module. In fact, by providing this template the system is also advising the user the a good first step in the definition of the new code module is the specification of the functional definition of that module.

Because of the use of a template, the IPE system is able to parse many of the portions of the functional description, so that knowledge base structures may be built which will permit searches to be made using many of the portions of the functional description. For example, the IPE can directly access and provide to the user, or to another tool within the IPE, the calling sequence of the subroutine and the description of the formal parameters as provided by the programmer.

This same information can be used to assist the programmer when the text of the code is being constructed. One type of assistance is that shown in Figure 2.6. Here all of the text shown has been automatically generated from the functional definition provided in the instantiated template of Figure 2.5. The formal parameter definition line has been generated using the calling sequence specification, and each of the formal parameter declarations in obtained from the description of the input parameters. The variable "mean_test_score" is declared as external and real based upon the information in the output description of the functional definition. Whenever a code module is changed, the IPE will rem-

```
PROCEDURE MEAN (SCORES, GROUP)
   BEGIN
     ARRAY SCORES [0:N,0:1]
     INTEGER GROUP
     EXTERNAL REAL MEAN_TEST_SCORE
     .
     .
     .
     END
```

Figure 2.6 Code Derived from the Functional Description

ind the user to update the associated documentation. Because of the close ties between the documentation and the code, the IPE will be able to identify whether or not the documentation has been updated since the last change in the associated code modules, and which documentation modules are associated with the altered code module.

Many other types of documentation are included in the IPE knowledge base. These include syntactic and segmented parses of the code, information about the "typical programming patterns" employed, descriptions of data structures, etc. Each of these types of documentation is defined and described in detail in the discussions about the types of information maintained in the knowledge base and the ways in which this information may be used which appear in later sections of this report.

It is important to note that great care must be taken with the way in which these documentation and assistance tools are used. Users should not be forced to provide information in a specific order or to approach the task of developing the software in a particular, rigid manner. Some users will be willing and pleased if the IPE guides them through much of the long list of tasks which must be performed in the development of a large software system. Other users have their own protocols for approaching these problems and will be greatly disturbed if forced to substantially change their ways. While changes can be advantageous in some cases, changes which do not have clear advantages will be resisted and will cause resentment and rejection of the editing tools.

Thus it is required that the IPE be able to utilize the knowledge of the various contexts of software development to assure that required information is provided and that the user is advised, when appropriate, about additional information which would be of substantial value. However, at the same time the IPE must be able to give the users the freedom to supply information in their own order and pace, and without too many stylistic or vocabulary restrictions. Some of these issues are discussed in more detail in the section below on user interfaces.

2.2.4. Semantic Analysis

The principle reasons for performing semantic analysis of the software being constructed is for completeness and consistency. A software system will be incomplete if some modules required to support the desired functions are not provided. It will be inconsistent if the same modules are employed in conflicting manners in different portions of the system. These problems are well understood, but few tools are provided in editors to assist the programmer in identifying or checking for such errors.

An example of consistency checking at the code level is shown in Figure 2.7. The types of consistency checks listed in that figure must be inferred from general knowledge about the programming language involved together with the specific knowledge of the code. Note that these consistency checks are to be treated as constraints upon the code which must be satisfied for the code to be considered correct. Such constraints will not catch all the errors which may occur in code, but they can catch many of the more common and well understood errors.

Some additional examples of high level consistency constraints are shown in Figure 2.8. While these statements would not be represented

S $\in$ SCORES IMPLIES SCORES IS A SET

S $\in$ SCORES IMPLIES S IS OF SAME TYPE AS
ELEMENTS OF SCORES

SUM + S IMPLIES SUM, S ARE OF NUMERIC TYPE
(INTEGER, REAL, . . .)

SUM + S IMPLIES SUM IS INITIALIZED EARLIER

Figure 2.7 Some Examples of Code Level Consistency Checking

A LOOP CONSISTS OF AN OPTIONAL INITIALIZATION, REQUIRED BODY, AND EXIT TESTS. EACH EXIT TEST MUST BE A BOOLEAN EXPRESSION OCCURRING WITHIN THE BODY.

A COLLECTION MAY BE ORDERED OR NOT, MAY ALLOW REPETITIONS OR NOT, MAY HAVE A SIZE ESTIMATE, MUST DEFINE THE PROTOTYPIC ELEMENT, AND MAY HAVE INSTANCES.

AN INPUT PRIMITIVE HAS AN INFORMATION STRUCTURE TO BE INPUT, SOURCE OF INPUT, OPTIONAL DATA FORMAT, PROMPT STRING TO BE OUTPUT, AND REPROMPT STRING TO BE OUTPUT IF DATA INPUT IS OF WRONG TYPE OR INCORRECT FORMAT.

Figure 2.8 Some Examples of High-Level Consistency Checks

within the IPE in English, it is clear that when represented in a form usable directly by the IPE they can be valuable constraints to enforce.

Similar types of constraints can be employed in the high level documentation of the code modules. For instance, checks can be made to assure that functions are called and employed properly, that all of the modules which are referenced in the functional definitions are available in code, that the functional descriptions are consistent with the code, and that supporting data specified in the functional definition is actually employed in the code.

2.2.5. Style Analysis

It is often the case that certain programming styles are more subject to inadvertent or difficult to detect errors. Text books about programming devote a great deal of effort to advising programmers about "good" programming styles. This includes advice about making systems modular, adding comments to the code, clearly describing any assumptions made, minimizing the use of "side-effects", etc. This advice is not always followed, and coding errors are sometimes the result.

Another type of tool which the IPE can provide is that of style analysis. This involves the specification and checking of constraints, much in the same manner as semantic analysis, which embody the style rules to be enforced. By making the style analyzer a tool of the IPE, the user can use all of the facilities of the IPE for altering code or documentation to conform to the style analysis guidelines. When appropriate, the IPE might be able to perform simple transformations to automatically correct style violations. In addition, the user would be provided with the ability to modify the application of the style rules, so that ones which are not essential and which conflict with the user's preferred style can be suppressed.

The following examples illustrate some of the possibilities for style guidelines. Some of these are general, while others are aimed specifically at the Ada programming language. For the purposes of this report, the *example style* guidelines are expressed in English; in the implemented IPE, they would be expressed in a more restricted language.

formatting guidelines:

- blanks should be used around assignment operators
- loop bodies should be indented

syntactic guidelines:

- do not assign to loop variables inside a loop
- do not use non-portable constants
- do not mix positional and name notation in the same parameter list

semantic guidelines:

- do not use variable before they are given a value
- use enumeration types instead of integer types if no arithmetic operations are performed
- do not use expressions in declarations that may have side effects

2.3. User Interface Considerations

Because the IPE is an interactive environment intended to directly support people doing software development, the way in which it interfaces to humans is quite important. The system must be easy to use, easy to understand, graceful and supportive, and tailorable to individual needs. In short, it must have a user oriented interface. While it was not the province of this project to investigate the needs for and characteristics of general user oriented interfaces, some examples of the types of interface activities can serve to establish guidelines for the later development of an IPE user interface.

The intent is for the user to be provided with a flexible and easy to use system, one which enables the full use of the power of the IPE tools. This requires several types of support. First, the user must be able to determine at all times exactly where he is and what has been left incomplete. This is illustrated by the following sequence of user activities:

1. The user begins with the description of the functional definition of module A.

2. The user stops at the point where module A invokes module B and begins to examine the documentation of module B (which had been previously defined).

3. Because the documentation is not sufficiently detailed, the user next pauses in the documentation of module B and requests a look at the code for module B, together with information about the data structures employed by that module.

4. At this point the user needs to refer briefly to the documentation for module A in order to determine more precisely the interaction between module A and module B.

Because the user needs to look back at precisely the point he left the development of the functional description of module A, he must be provided with a "map" of all of the steps in his path since he began the recursive investigation of the modules. Such a map will enable him to determine not only where he is now, but how he got there and how to reference earlier steps in the path.

A second type of user interface support is a uniform means by which all the portions of the extended program definition (code, documentation, knowledge base representations) can be accessed. In other words, as our user moves from the specification of the functional definition of a module to the specification of the code text to the specification of the algorithm in terms of the relevant Typical Programming Patterns (TPP's) (described further below), he should not be required to substantially shift the means by which search requests are expressed or new information is provided. The fact that each of these segments of the knowledge base are represented differently must not be visible to the user, only to the internal processing of the IPE.

The third major aspect of the user interface is its use of the programming context model to guide the nature of the interactions with the user. While the IPE executive uses the context model to determine the types of activities which must be undertaken and the relationships among those activities, the interface must use this context knowledge to determine the ways in which the information should be presented to the user. For example, if the user is in the process of defining the algorithms to be used by a module, he will need access to the TPP descriptions, as these are the generic algorithms known to the IPE. Part of the information which the user will need to provide to the IPE knowledge base (so that it is available for later search requests) is the correspondence between lines of code in the program text and the logical portions of the algorithm as defined by the TPP. It is essential that the facilities provided to allow the user to describe the correspondences be both easy to use and natural. One way would be to use a split screen, showing the TPP and the program text side by side, and enable the user to use a cursor controlled by an electronic tablet or a pointing device so that the desired correspondence can be specified by drawing arrows from the TPP component to boxes around the corresponding code sections.

The fourth major characteristic of the user interface is that it must be able to be customized to the needs and approach of each individual. This means that there must be some form of user model used by the interface, and that this model must be able to be defined and updated by the user. This will enable user A to define one set of built-in searches to be associated with a function key on a keyboard, while user B has a different set. This type of customization is essential if users are to be supported with the short-cuts that will make the use of the IPE effective and efficient. Repeated use of long and cumbersome methods for specifying searches done many times each session will greatly impede user utility and user acceptance of the IPE tools.

Other issues beyond those we have been able to identify in the course of this limited study and the implementation of the experimental system described in this report are likely to extend the list of capabilities required for the IPE user interface. However, we believe that all of the considerations will fall within the three principal areas discussed in the sections below: input methods, output methods, and command methods.

2.3.1. Input Methods

There are a number of alternative input methods available for the IPE. Traditionally interactive systems have relied upon alphanumeric keyboards at which users type. This information tended to be sent directly to the underlying programs, with the interface performing little translation or transformation of the ASCII character strings. Sometime simple changes such as the breaking of lines longer than 80 characters are made by the interface.

The two parts of a more robust and powerful input interface are the use of alternative input devices and the availability of more powerful transformation tools. Alternative input devices include voice recognition devices, electronic tablets upon which users may hand print text, and a variety of touch sensitive and other pointing devices. Given these inputs, the interface may also be responsible for transforming the user's input into a form more suitable for the IPE to process directly. For example, spelling errors can be corrected, ambiguities resolved, and free-form inputs forced into more structures formats. In the early development of an IPE it would be appropriate to support inputs from a keyboard and simple pointing devices, such as a "mouse". As the sophistication of the IPE system grows, other types of pointing devices, voice directed input forms, or even movement following devices might be employed.

2.3.2. Output Methods

The means by which computer systems have traditionally communicated with the users have also been severely restricted. Usually output is in the form of alphanumeric text on either CRT terminals or hardcopy devices. Users have little choice about the format of the output, the amount of output provided as a single unit, or the structuring of that output. Little attention has been paid to whether or not the information will be useful to specific classes of users, whether the structure of the information makes it easy or difficult to interpret, or whether the user is being overloaded with data.

Alternative output media include graphics (monochrome or color), photographs, animation, sound, and combinations of these with standard text. In all cases, however, it is essential that the system have the ability to format the information presentation in a manner which is both intelligible and pleasing to the user. Thus we must make a careful combination of data with high information content and presentation with good human factor considerations. The most likely candidates for initial versions of the IPE would be multiple text fonts together with bit-map graphic presentations of text with multiple window capabilities. Later improvements might include drawings (for flow charts and similar graphical presentations) and even animated graphics for showing the dynamic changes of data structures as a part of debugging tools and tools for showing program execution as a part of a comprehension context.

2.3.3. Command Methods

In conjunction with the many alternative input methods and devices, it is possible to give the users many alternatives for the means to instruct the system with regard to the actions desired by the user. For example, in addition to the usual types of alphanumeric commands, users may utilize menus from which possible commands are selected using either numbers or various pointing devices. Some simple types of commands might also be given verbally, or commands might be made in the form of typed English sentences or fragments.

The choice of methods again is dependent upon the preferences of the users, but it is clear that the user interface should provide a range of alternatives. This will enable individual users to tailor the system to best suit their own modes of operation. In line with the current types of user command approaches which seem to be favored in many commercial products, the IPE should probably provide the facilities for controlling IPE activities using simple command words and characters, pre-programmed function keys, and hierarchies of command menus for use with pointing devices.

3. The Extended Program Model

The Extended Program Model (EPM) is a tool for representing, accessing and manipulating programs in a sophisticated way. It accomplishes this task b/ defining a vocabulary for discussing programs which uses terms that are much closer to the ones which programmer's naturally employ. The EPM also contains a knowledge base that documents the composition of particular programs. This information forms the backbone of systems such as the IPE, which exhibit a deep understanding of the organizational structure of code.

The EPM's knowledge base explicitly represents the physical and the functional structures that are present in a given program. This information is organized into a collection of knowledge sources which describe the program from a number of different views. So, for example, the EPM contains an image of the program as text, and as a syntactic object with an associated parse tree. At the same time, it maintains a data and control flow analysis, as well as a breakdown of the code into the standard programming actions which it employs. The EPM also contains knowledge from more semantic domains; it associates program structures with frames representing the user's intent in his application domain, and it allows documentation templates to be attached to any of the features described above.

As a whole, the EPM can be thought of as a data base management system for maintaining code. It provides a search language for accessing its knowledge, a manipulation facility for performing updates, as well as a set of semantic integrity and consistency constraints for monitoring the validity of the data it contains.

The EPM is implemented in terms of two major subsystems: the program reference language (PRL), and the program structures data base (PSDB) which maintains the knowledge referred to above. Please see figure 3.1.

3.1. The Program Reference Language

The PRL is a tool for flexibly accessing the interesting portions of programs. It is envisioned as a query language for examining code, analogous to an information retrieval system for scanning text. The PRL's goal is to take a description of a program fragment, compare it against the knowledge base of the structures present within the program, and return the matches that are found. The PRL provides a formal reference language for stating search requests. It implements those requests by translating them into a series of primitive search operations which are defined on the underlying knowledge base.

The PRL is motivated by the observation that programmers are often in the situation of dealing with partial information concerning code. This occurs in the process of editing programs which are too large to remember explicitly, in the act of understanding code which have never been seen before, or in the process of completing partially implemented designs. In each situation, the programmer typically knows what he wants to see next, but he does not have its exact description. The PRL bridges this gap by employing a generalized vocabulary for referencing code.

Search requests come in a variety of forms. At a simple level, the programmer might want to locate the code fragment which contains the

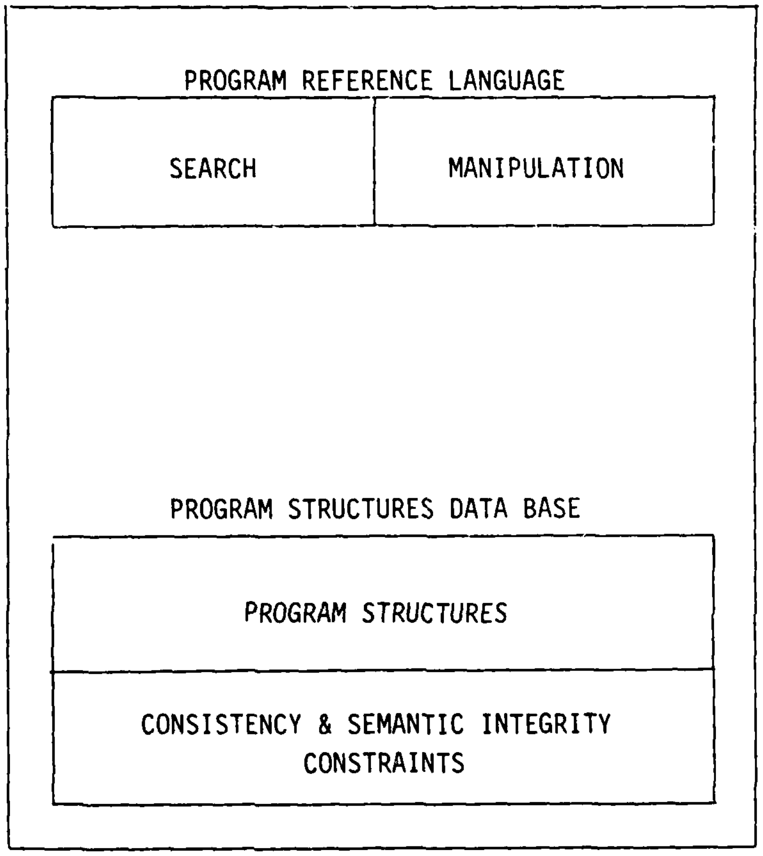

PROGRAM REFERENCE LANGUAGE
SEARCH
MANIPULATION
PROGRAM STRUCTURES DATA BASE
PROGRAM STRUCTURES
CONSISTENCY & SEMANTIC INTEGRITY CONSTRAINTS

Figure 3.1 Extended Program Model (EPM)

string "Procedure Sum:". This corresponds to a standard textual search operation available in current editors. In a more complex request, the user might look for the "procedure declaration with procedure name Sum". (The quotations are for emphasis; this request would be specified in a more formal language, not in English.) This query references the syntactic structure of the program. Similarly, the programmer could search for the place where an action from the domain of programming, such as summation is employed. In order to use this vocabulary, the search system needs to have access to definitions for typical programming actions that the programmer might employ. Finally, at a very complex level, the programmer should be able to locate a fragment by combining requests operating in any of these domains. For example, the statement, "show me the loop that computes the sum of the test scores" references the intent behind a program fragment (since test scores are a concept from an application domain), a typical programming action (summation), and a syntactic construct, namely a for-loop or while-loop. These searches are all within the domain of the mechanism we are describing. (In chapter 4, we discuss the implementation of a system which answers this exact request.)

The manipulation portion of the PRL provides a mechanism for incorporating new knowledge into the program structures data base. This task involves two types of activities: specifying the additions to be made, and modifying the contents of the appropriate knowledge bases. The manipulation system employs the vocabulary defined by the PRL. So, for example, if the user wants to state that a region of his program implements a set insertion, he would specify the addition by naming the actions of the set insertion (a membership test, a splice-in operation and perhaps an ordering test), and then relate them to the corresponding sections of code. The manipulation system would then build the appropriate set insertion template, and link it in to the remainder of the structures in the knowledge base. The system shields the user from the details involved.

The second function of the manipulation system is to monitor the consistency of the program structure data base. This is primarily a matter of keeping the various portions of the system up to date. For example, if the user edits a routine, the program text changes, and all the documentation which references it has to be examined in turn. Some parts of this function can be implemented automatically. For example, there are parsers which can produce a syntax tree given program text. However, this function will often require interaction with the user. As a case in point, no automated system is able to recognize when documentation is made invalid. In situations like this, the goal of the manipulation system is to notice which portions of the data base have to be re-examined, and then to bring them to the attention of the programmer so that they can be modified.

The following sections describe in more detail the EPM's knowledge base, its semantic integrity and consistency constraints, and then the search and manipulation capabilities which the PRL provides. Up until this time, we have devoted the major portion of our effort to the task of refining the knowledge in the PSDB. This research has progressed to the point that we are currently implementing a portion of that knowledge base, together with a primitive search facility for accessing the data it contains. The system is called the PRLI (program reference language implementation), and is presented in chapter 4. The formal reference language which uses this information has not yet been defined.

3.2. The Program Knowledge Base

One of our earliest observations was that programs are structured objects whose organization is reflected at many different levels of detail. They contain textual and syntactic information, data and control flow, iterations and recursions, and other stereotyped behaviors which are implemented by simple program language constructs. At a more abstract level, programs, as objects, reflect the intentions of the programmer. Regions of code can implement actions from the user's domain of application, or programming cliches, such as set abstractions, list insertions, and hash tables instead. In order to represent these types of structures, the PRL's knowledge base has to contain information from a variety of sources.

We have constructed this data base in terms of a hierarchy of knowledge sources which reflect the transition from a syntactic, to a more intention-oriented analysis of code. (Please see figure 3.2) It includes textual, syntactic and semantic representations, as well as a domain independent representation for documenting regions of code. Each of these analyses brings out a particular aspect of program organization, and emphasizes some set of manipulations which are important to perform. For the purposes of the PRL, we are considering these viewpoints to be abstract data types which facilitate different sorts of retrieval operations.

The data in these different abstractions is also strongly interdependent. For example, a program can be viewed as a list insertion, but it is still implemented in terms of syntactic constructs. Similarly, the text of a routine may be the only compilable version of the program, but it also provides a statement of many more abstract intentions. The point is that the program structures data base needs to support all of these viewpoints simultaneously. Its components form a richly interconnected set of knowledge sources, as shown in figure 3.3.

In the following sections, we describe each of these program viewpoints in turn, but from the vantage point of the PRL. In this context, we first discuss the type of information the knowledge bases contain, and then describe their effect in terms of the search operations which they facilitate.

3.2.1. Program Text

The text string representation is included within the PRL in order to give it the standard search capabilities present in typical editors. It is a low-level approach, concerned with words and delimiters as opposed to the semantics of programs, but it is important nonetheless. Examples of search requests on text are, "find the next occurrence of the string, ´procedure:´", "move forward 3 words", and "move back 6 lines". This can be extended to include pattern matching capabilities, where the search form, "abs*" would match the words "absolute-value", "abstract", and "abscissa".

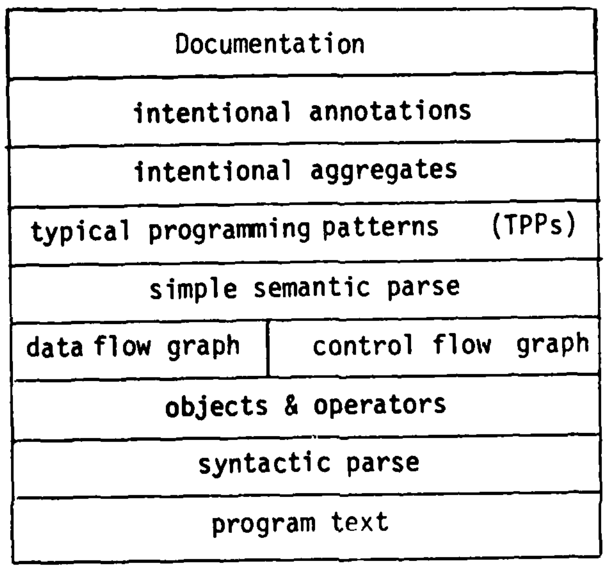


Figure 3.2 Hierarchy of Program Structures in PRL

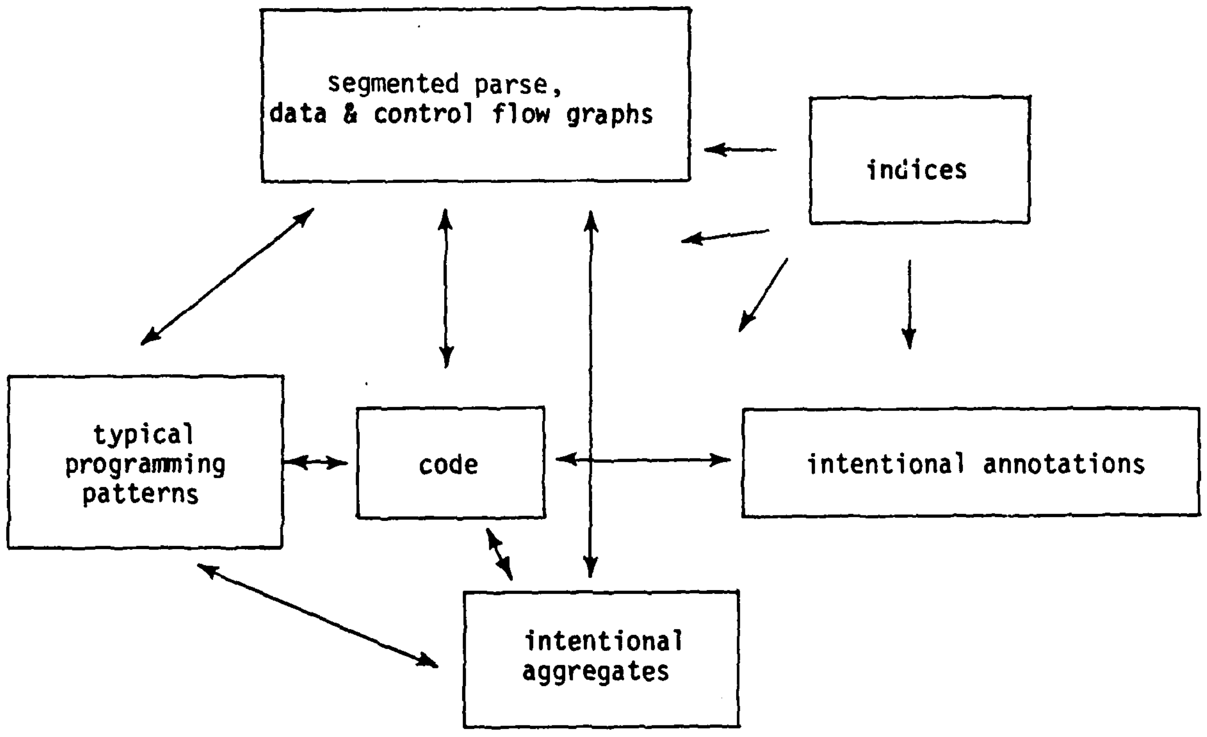

segmented parse, data & control flow graphs
indices
typical programming patterns
code
intentional annotations
intentional aggregates

Figure 3.3 Interconnectivity of PRL Structures

3.2.2. Syntax

Syntax refers to the rules of grammar for a particular programming language. It can be thought of as a series of constraints which build legal programs from textual objects. In ADA, this means knowing that "begin"s and "end"s come in pairs, that "restricted" and "privatc" are optional designations within abstract data types, and that "accept" is a valid keyword only within concurrent tasks of the message-handling type, etc. We are not planning to represent this information in terms of a parser that examines a naked text string; we expect to model syntactic forms with a network of templates for specific syntactic structures. In this way, a statement equivalent to "find the implementation for the sort operation on lists" would translate into a search for the keyword "sort" through the "operations" slot of the "private" part of the template representing the list abstraction. This approach takes one branch of a time-space tradeoff, but is consistent with our philosophy of explicitly representing program structures.

3.2.3. Objects and Operators

Objects and operators form the vocabulary for the entities which programs manipulate and for the manipulations which can be performed. They are the primitive constructs which carry the semantics of programs. In the perspective taken by the PRL, objects and operators include both the built-in data abstractions provided by most programming languages (integers and integer arithmetic) and the user defined abstractions available in strongly typed languages such as ADA.

Objects and operators are not relations like data flow which facilitate certain types of search requests. They are the nouns which programmers refer to in the process of forming questions. So, for example, it is important for the PRL to know that queues are objects in order to be able to locate "the routine which operates on queues". In building more complex structures, data abstractions provide a place to attach higher-level semantic constraints. For example, legal objects and I/O relations can be given mathematical descriptions, and since abstractions often form the organizational framework of large systems, they tend to fulfill specific purposes which can be captured with annotations. These descriptions could be used by the PRL.

3.2.4. Data and Control Flow

Data flow and control flow are two of the fundamental components of program structure. They exist in programs regardless of their source language or any more global approach which they employ (meaning message passing, parallel computing, or procedural embedding, to name a few). Data flow refers to the connectivity between the operators which produce and consume data. Control flow identifies the pathways through the operators which execution might involve. Control flow diagrams branch at conditionals.

Data flow analysis is a powerful technique for resolving programs into functionally related parts. It has been used by the Programmer's Apprentice project at MIT to motivate a large part of their effort in program understanding [Rich-81]. There are two key features of a data flow analysis. First, it associates operations which may have been widely separated in the original text, but which bear a functional relationship to one another. Intuitively, this corresponds to the notion that any operator which uses data that another produces is working with it towards some unified goal. Secondly, a data flow analysis simplifies programs by abstracting away the method in which data connectivity is implemented. For example, it makes no difference if procedure calls, global variables, or a series of assignment statements connect two operators; each method reduces to the same data flow.

Control flow analyses have analogous properties: they associate operators that come from disparate sections of the program text, and they abstract away the details of control flow implementations. For example, conditionals and "case" and "go to" statements which implement the same fundamental control sequence map into a single format. As a consequence, it is more general to represent higher-level structures as patterns of data and control flow than as patterns in the original text. This representation facilitates the recognition process, because it maps programs into a more regular form.

In the context of the PRL, a primitive data flow analysis will make it possible to locate code fragments which produce and consume data, or which input it or operate on it in any way. Examples of questions of this type are "who uses this variable?" or, more potently, "who uses the data that this variable represents?". (Simple cross-referencing tools handle the first, but not the second.) A control flow analysis provides the ability to locate code by its position in an execution path. Examples here include, "who calls this routine?", and "what are the set of modules which this routine can ultimately invoke?". It should be noted that at this level of sophistication, we are only examining the connectivity within a program, and not the reasons why particular patterns of connectivity exist at all.

The data and control flow analyses described above can be united into a single knowledge source that captures the low level behavior of a program. In this view, called a segmented parse, patterns of data and control flow are used to define a vocabulary of simple actions. So, for example, iterations are decomposed into a a set of roles which identify the subfunctions of a loop. In the breakdown we are using (formalized in [Waters-78]), loops contain generators, filters, terminators, and augmentations. Generators are segments which produce a sequence of values; they correspond to the "i = i + 1" part of a loop, or the process of "cdr"-ing down a list in LISP. Filters restrict that sequence of values. A terminator is like a filter, except that it has the additional potential to stop execution of the loop; so for example, the statement "if i > n then go to Exit" restricts the values of "i" seen past that point in the loop to those less than "n". An augmentation consumes values and produces results. For example, the phrase, "sum = sum + A[i]" implements an augmentation.

3.2.5. The Segmented Parse

Taken together, these fragments produce a complete parse of the behaviors in a loop. For example, the procedure shown in figure 3.4 computes a thing called the "sum", using statements which fulfill the roles discussed above. (This program is written in a PASCAL-like dialect, where "go to"s are used only to bring out the occurrence of the loop. It is important to notice that the input language is unimportant to the analysis; the loop would be parsed identically if it were written in any other language.) What is missing from this analysis is any expression of the purpose of the code, namely that it adds together the positive elements within the first 10 positions of the array. Nevertheless, this taxonomy is quite useful to the PRL; it provides a method for locating portions of code according to the roles they fulfill within a larger context. It raises the level of the vocabulary available for discussing programs.

These loop primitives (and by extension, all iterations in code) can be represented as patterns of control and data flow, although the process of automatically generating that breakdown can be computationally expensive. In the Programmer's Apprentice project it is accomplished in two phases: first, a program is completely resolved into its data and control flow. Next, this information is segmented or associated according to the data and control flow relationships which define the different roles. In the PRL research, we are primarily concerned with examining, as opposed to creating such representations, so we assume that these automatic analysis procedures have already been performed. In terms of our implementations of the PRL and the IPE, this means that we will manually enter the data structures in their entirety, and focus instead on the search procedures that examine them.

A segmented parse of the program in figure 3.4 is presented in figure 3.5. This diagram is complicated because it explicitly represents all of the data and control flow relationships which are usually implicit in program text. However, we will not discuss it in all of its detail (see, e.g., [Shapiro-81]). By way of introduction, the solid

```
let maxsize = 10;
let sum = arraysum (A, maxsize);
let average = sum/maxsize;
        •
        •
        •
real procedure arraysum (A: real array, n: integer)
   begin
       integer i;
       real sum;
       let i = 1;
       let sum = 0;
       Loop:  if i > n then go to Exit;
              sum = sum + A [i];
              i = i + 1;
              go to Loop;
       Exit:  return (sum);
   end;
```

Figure 3.4 Example Code

lines represent data flow, and the dashed lines show control. The boxes correspond to the types of entities which form the semantic parse (generators, filters, etc.). The roles they fulfill are labeled on their upper left-hand sides. The boxes with no further substructure correspond to primitive operations in the code, such as "+", or ">". The elements labeled "R" and "J" are not important to understand here, beyond the fact that "R" stands for a recursive instance of the box which contains it, and the "J" element joins data or control flow when it may come from one of two places as the result of a program branch. The data flow line labeled "i", which connects the generator and terminator bodies, should be thought of as transmitting a set of data; it provides the value of "i" at each iteration of the generator body.

The important features of the diagram are as follows: first, it is segmented into named roles. There is a generator, a terminator and an augmentation which have substructures that are also named parts. In some cases there are required parts (a terminator must contain a predicate), and in others the substructure is optional (the initialization segments do not have to be present). The segments also associate related actions; the array reference, the sum initialization and the summation operation all occur inside the augmentation, as opposed to disparate sections of the code. The next point is that the diagram presents only the essential interconnections between operators in terms of data and control flow, but not the method used in the program to implement them. There is no reference to temporary variables or "go to" statements in the figure. The final point is that the segmented parse is language independent. It does not contain any features which tie it

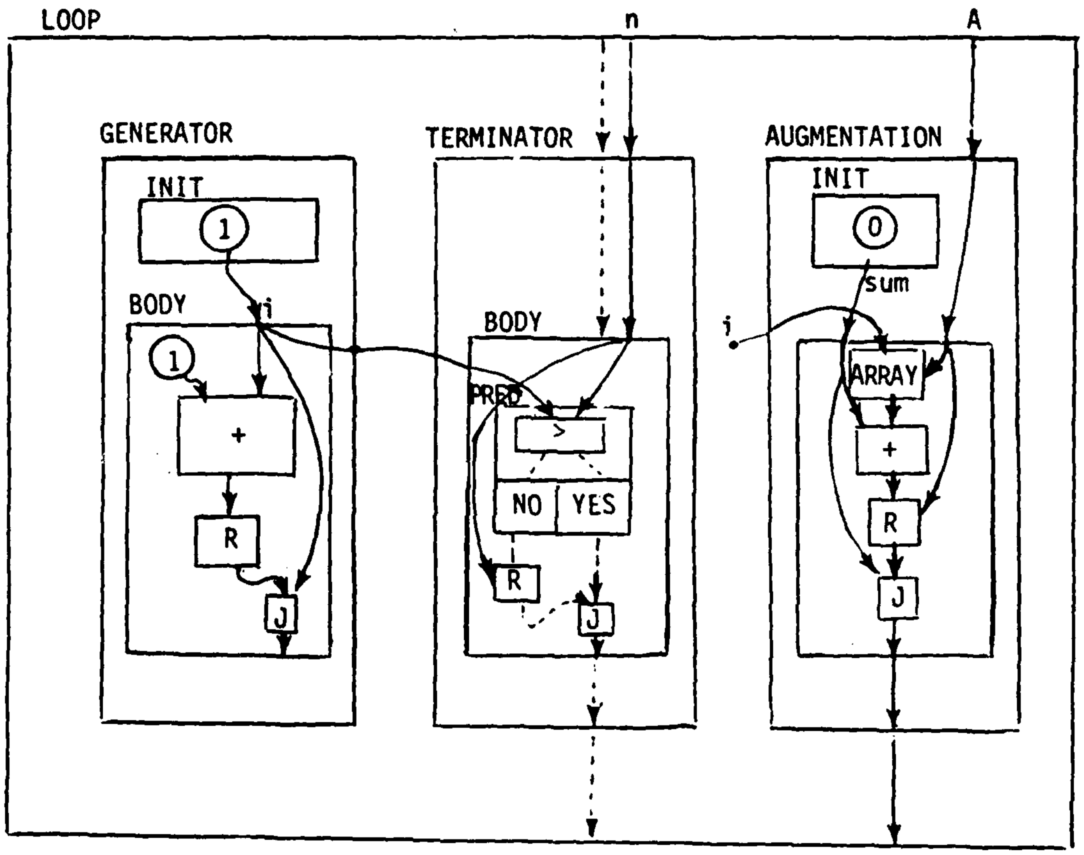


Figure 3.5 Segmented Parse

to a particular source language. An example of how the PRL can use these diagrams is presented in a following section.

3.2.6. Typical Programming Patterns

The taxonomy that has been discussed up until this point primarily captures information about the form of programs as opposed to their meaning. The only semantic elements we have introduced described the substructure of built-in entities such as loops. In the next, more abstract viewpoint, we consider programs to be built of objects with stereotyped purposes. These are called typical programming patterns, or cliches. Examples of cliches include variable interchanges, list insertions, hash table abstractions, and queue and process strategies (which are used heavily in simulations and event-oriented programming). These algorithms are the tools employed by every expert programmer; they occur in programs of any size.

Cliches are represented in terms of the segmented parse described above. For example, the act of summing up a sequence of elements can be represented as the subset of figure 3.5 which is required in every summation loop. This is presented in figure 3.6. Here, it is not important to know what order the indices into the sequence are generated in, or how, only that they are produced, and then consumed by the summation calculation. Similarly, a termination test must be present, and there must be an augmentation which acts as the accumulator of the sum. The vocabulary of the semantic parse indicates that there are optional segments as well.

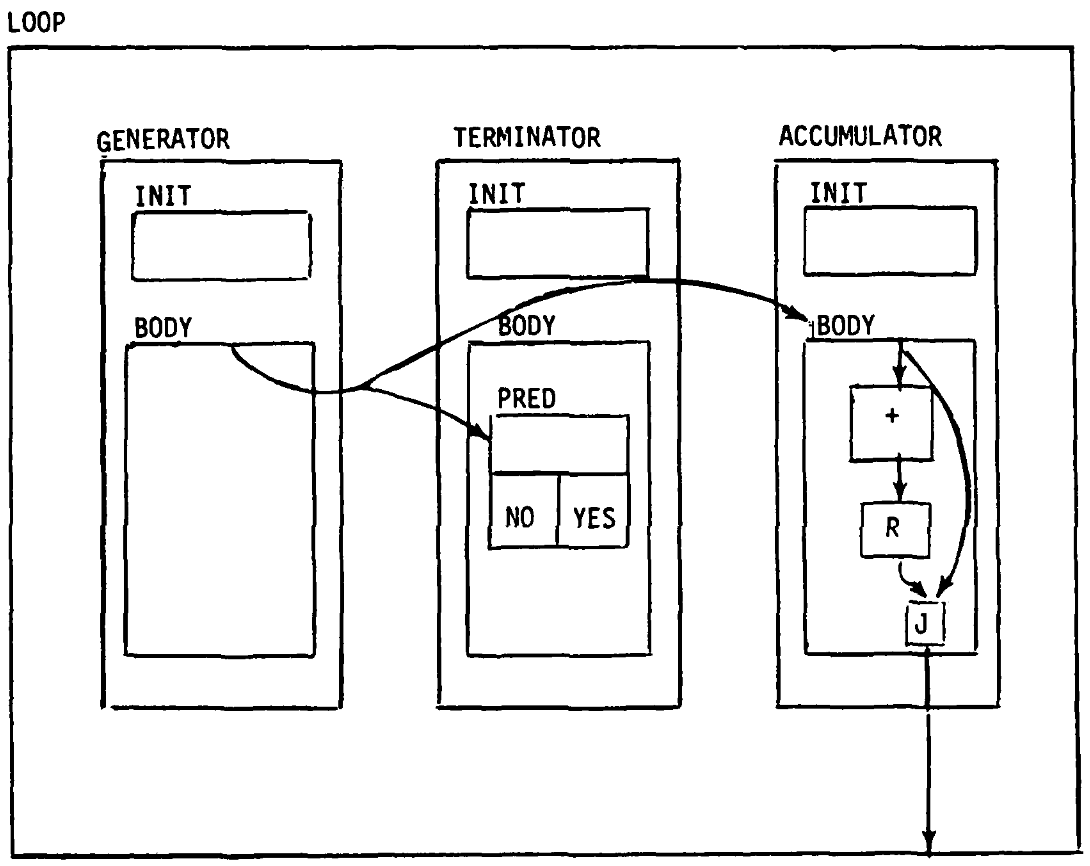


Figure 3.6 Summation Loop TPP

3.2.7. Intentional Aggregates

Intentional aggregates provide a method of associating the user's intentions for a program with the sections of the code that implement them. Their purpose is to capture pragmatic knowledge, meaning facts from the domain of application governing the code. Intentional aggregates are structured objects (represented as frames here) which have no innate semantics to a system like the PRL. However, they do associate program fragments with named keywords that can be used to search for those regions of code. So for example, the program for the averaging computation shown in figure 3.4 corresponds to the intentional aggregate in figure 3.7.

This aggregate is composed a collection of slots which have named facets that contain values. Here, the slots are "name", "purpose", "parts" and "mappings", although other ones may be present as well. The facets are "sum computation", "maxsize", etc. The value of a facet may be a primitive descriptor (such as a string), an instance of a typical programming pattern, or simply a pointer to a piece of code with an annotation such as "this computes an average" attached. Intentional aggregates can reference structures defined in any of the knowledge sources described so far. The aggregates may also represent noncontiguous portions of code.

3.2.8. Intentional Annotations

Intentional annotations provide a means for representing a fixed set of facts which are generally useful for identifying regions of code.

NAME: intelligence-test-statistics #1

PURPOSE: compute the average of the IQ test values

PARTS:

sum computation	TPP-summation #12 (Figure 3.6)
average computation	"let average = sum/maxsize" (Figure 3.4)

MAPPINGS:

"A"	the array of test scores
"maxsize"	number of scores in the array of test scores
"sum"	sum of the test scores
"average"	average of the test scores
"n"	number of scores to be summed
"i"	index into the array of test scores

Figure 3.7 Intentional Aggregate

They are structured documentation packages that can be attached to modules and files, which record comments about the goals, assumptions, expectations and requirements of a program section. In addition, these annotations have slots for the more detailed facts about code; for example, they identify the author, creation date, modification history, and bugs which are known to remain in a particular module.

Intentional annotations provide two important types of advantages to the programmer. First, they can help regulate the process of documenting code. So, for example, whenever the programmer defines a new module, an intelligent editing system will be able to prompt him for specific information that should be associated with that code. Second, the PRL will be able to retrieve modules and files according to the data in these fields. Since the templates are predefined, simple indexing mechanisms can be employed. Several examples of intentional annotations were presented in the previous chapter. Please see the module description and function definition templates presented in figures 2.3 and 2.5.

3.2.9. Documentation

Intentional aggregates and annotations are subsets of a generalized form of documentation which the EPM provides. In this knowledge source, we allow the user to associate textual comments with any of the program features already described. So, for example, the user can document the data flow in a particular module (saying why that input output relationship occurs), justify his use of particular TPPs, or identify why particular syntactic features were employed. This system takes advantage of the EPM's partitioning of program knowledge to classify comments in useful ways.

We have not explored the utility of this abstraction beyond a preliminary stage, but it addresses an important issue in the design of program understanding systems (such as the PRL/IPE). In specific, in order to mirror the ways in which people view programs, these systems have to extend their representation capabilities into increasingly more semantic domains. The EPM attempts to make some of this organization explicit, by providing the segmented parse, TPP and annotation formats already described. This documentation knowledge source is aimed at capturing some of the less visible semantics inherent in the free form text associated with code.

In terms of search, we have already discussed simple methods which can be used if documentation is treated as free form text. However, it is also possible to apply more knowledge based tools. AI&DS has implemented a system for performing rule-based information retrieval, which could also be employed here.

3.3. Semantic Integrity and Consistency Constraints

When the EPM is viewed as a data base management system for representing programs, several new difficulties emerge. First, there is a semantic integrity problem, meaning that the data in the knowledge base must correspond to a valid program. Second, there is a consistency issue, which implies that all of the EPM's overlapping knowledge structures must be kept up to date.

The EPM's consistency mechanism primarily identifies the consequences of particular additions to the knowledge base. It accomplishes this task by virtue of the explicit interconnections that the knowledge sources contain. (See figure 3.3) So, for example, if the programmer modifies a particular module via the textual representation, the syntax, and segmented parse data will have to change in turn. In addition, it is also possible that the code has been modified to the point where the TPP, annotation and documentation segments that reference that code are no longer applicable. In this case, a large number of changes will have to occur.

The consistency mechanism in the EPM will be able to automatically update some of those representations; for example, there are many existing tools for syntactically parsing a given piece of text. A segmented parse can be created automatically as well (see [Waters 1978]). However, in the remaining cases (in the context of current, or near term technology), it will only be possible to alert the user when particular information may have to change.

The EPM attacks the integrity issue by defining a collection of rational form constraints which operate on the knowledge sources discussed in the previous section. (Here, the term "rational form" refers to the fact that any code which violates the form is nonsensical in some way.) We have identified three types of rational forms, which apply to the syntax, object and operators, and data and control flow domains.

Rational forms are meant to be more powerful than a type of validity check which examines only the format of the data in a given frame. This kind of error can arise whenever the user is allowed to edit a formalized representation. For example, he could insert an illegal statement into the syntactic parse (an "end" with no opposite "begin"), or he could delete a necessary role within a segmented parse frame, e.g., a loop without a generator is not well defined. These are all examples of simple constraints, along the lines of syntactic checks on each type of representation. In the remainder of this section, we describe deeper forms of limitations, which get at the meaning of the objects in each domain.

3.3.1. Rational Syntactic Forms

Rational syntactic forms are compound syntactic structures which must occur as a whole in any program, if they occur at all. Examples include the fact that procedures must be called with the appropriate numbers of parameters, and that the I/O specifications of procedures in the visible portions of ADA modules must agree with those in their implementations. This notion can be extended to capture the fact that syntactic sugar is required in order to support particular actions. For example, the ADA "accept" statement must always occur in the following context:

```
TASK foo IS
     ENTRY PROCEDURE bar
     END;

TASK BODY foo IS
     ENTRY PROCEDURE bar
             BEGIN
                    ACCEPT write
             END;
     END;
```

In the context of program development in the IPE, just typing the form "ACCEPT write" would produce the entire pattern described above.

3.3.2. Rational Object and Operator Forms

A violation of rational form, in the context of operators and objects, refers to places where their expectations are not satisfied. The best example of this comes from type checking; since every operator categorizes the type of its output, and places restrictions on the types of its inputs ("+" is defined on integers or reals but not on strings), it is possible to chain calling sequences together to determine if a type mismatch has occurred. This includes type checking on user defined data types, which are provided in structured languages such as ADA. Data abstractions typically contain other annotations on the objects they manipulate which define constraints on their contents. For example, ADA provides for range restrictions on the elements of arrays, or a programmer may define a negative bank account balance to be impossible. Some of these violations can be caught at compile time with the aid of advanced techniques such as symbolic evaluation or special-purpose verification, but for the most part they represent errors which can only be caught at run time, when objects actually receive values. As such, they are of limited utility to the PRL, which is currently defined to operate only on static, as opposed to dynamic structures.

3.3.3. Rational Forms in Control and Data Flow

Rational forms in control and data flow delimit reasonable constructs built of those primitives. For example, it should always be possible to reach every piece of code within a program. Similarly, endless loops (loops with no possibility of termination) are not allowed. An example of a rational form in data flow is the fact that "dead ends", meaning places where data is produced but never examined, are not allowable.

In this viewpoint, rational forms are not necessarily ironclad rules; they can be violated in sufficiently unusual contexts. For example, at the core of every computer system there is a loop which runs continuously until the machine goes down. In systems which contain self-modifying code, control flow paths which theoretically exist might not be possible to observe (this is only a dubious violation of the rational-form constraint). Concerning data flow, it is occasionally useful to call a routine for its side effect, and never examine the value that it returns. In any case, these constraints form useful validity tests on programs. The IPE can use them in the positive sense by searching for code sections which violate the rational forms, for example, by locating places where data is generated but not consumed.

3.4. The Manipulation Mechanism

The manipulation system is the least well defined portion of the EPM. This situation was expected, in the sense that we had to identify and represent the important kinds of program information before we could think in terms of altering the data structures which we defined.

The system however, performs two functions. First, it provides a mechanism, analogous to the PRL's search component , which can implement manipulation requests in terms of the operations which the individual knowledge sources provide. Second, it applies the semantic consistency procedures which alert the user about the effects of a change. (This mechanism was described in the previous section, and will not be discussed further here).

The manipulation system relies on the vocabulary provided by the PRL. For an example, please see figure 2.4. The discussion in that chapter identifies an editing transformation which the IPE might request, for translating "while" statements into their equivalent for-loops. It is important to recognize where the division of labor occurs. In this case, the IPE contains a rule stating the transformation in the reference language. The manipulation system is responsible for carrying it out, using its knowledge of the detailed operations available for building segmented parse structures and syntax trees.

The transformation could be accomplished in the following way. First, the "initialization" which precedes the while-loop (taken from the segmented parse domain) becomes the "iteration variable" and "initialization" of the syntactic template for the for-loop. Next, the bump step, which corresponds to the "operator" in the "loop generator" becomes the integer associated with the "step" node of the syntactic tree. Similarly, the "termination condition" of the while-loop is extracted, and its upper limit is entered into the "termination clause" of the for-loop representation. In this form, the transformation involves a mixture of information from several EPM domains. (We should also note that the segmented parse structures for those two loops are nearly identical. A small amount of knowledge about automatic coding in that domain would allow the two forms to be created equally.)

3.5. A Search Example

The scenario discussed in this section shows how the data structures maintained by the PRL can be used to answer a complicated search request. This search process coordinates a wide variety of information, and takes advantage of the rich connectivity within the PRL's knowledge base. However, our example does not display the exact order of events in the solution process; since our understanding of how to control the search is still incomplete, we discuss one plausible path through the data structures involved.

The question posed to the PRL is, "find the initializations of the loop which computes the sum of the test scores". (This is the English equivalent of the PRL request.) It is processed against the averaging computation represented by the code shown in figure 3.4. At the outset, the question can be seen to require both high-level and detailed information about code. For example, the terms "initialization" and "loop" have to be meaningful to the PRL. They refer to semantic and syntactic information from the hierarchy in figure 3.2. The concepts of "sum" and "sum of the test scores" access more pragmatic knowledge. To use this information, the PRL must know about typical programming patterns, and application-specific concepts within the code.

We envision the search process to proceed in the following manner. (Figures 3.4 through 3.9 will be important. Figure 3.10 summarizes the steps involved.) It is reasonable to assume that, as a starting point, the PRL has an index of all of the loops contained in the program. Each of these corresponds to a segmented parse, of the kind described in figure 3.5. In this example, the PRL can limit the number of loops which it considers further by applying constraints obtained from the user's request. Here, the relevant structures must contain initializations.

PRL Query: Find the initializations of the loop which computes the sum of the test scores.

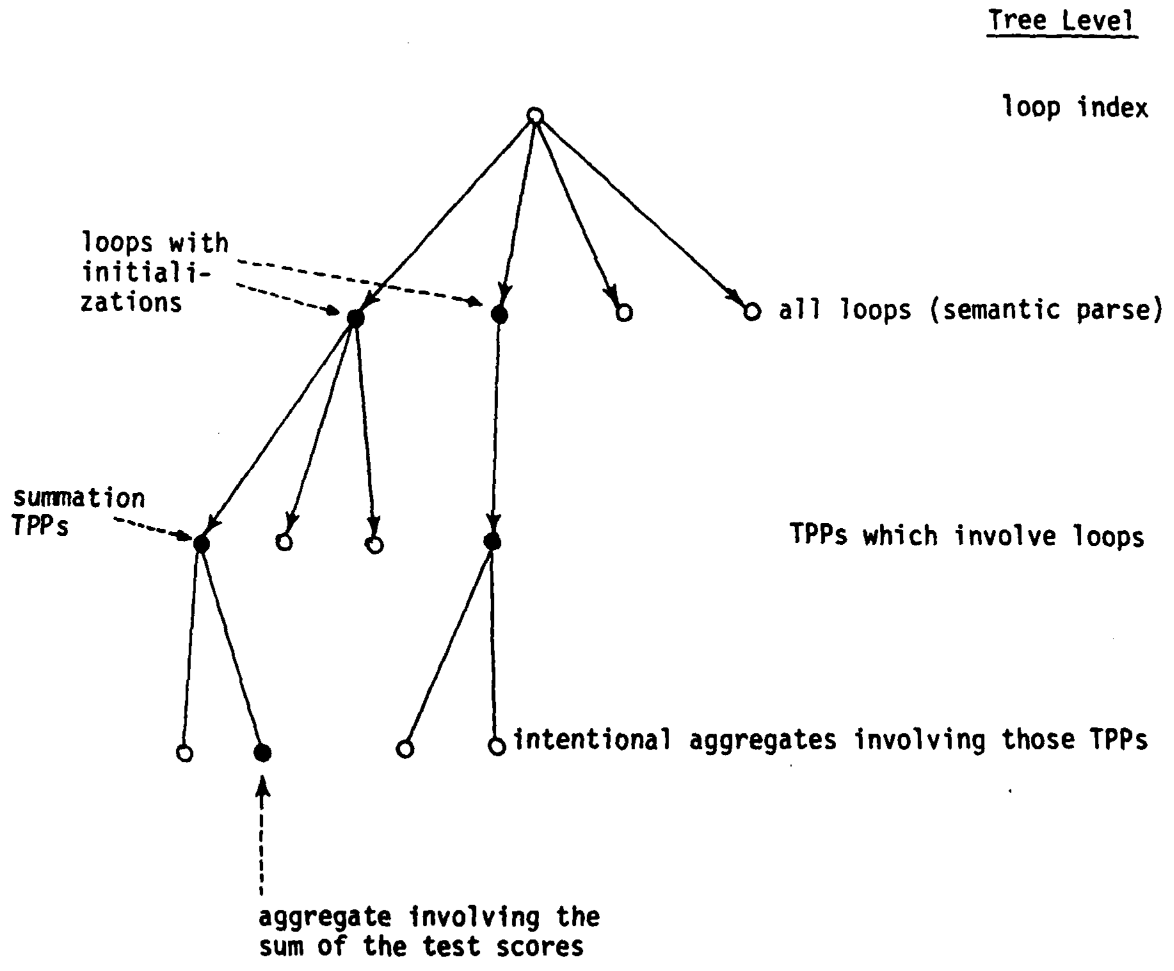

Tree Level
loop index
loops with initializations
all loops (semantic parse)
summation TPPs
TPPs which involve loops
intentional aggregates involving those TPPs
aggregate involving the sum of the test scores

Figure 3.8 Search Example

```
let maxsize = 10;
let sum = arraysum (A, maxsize);
let average = sum/maxsize;
        ●
        ●
        ●
real procedure arraysum (A: real array, n: integer)
   begin
       integer i;
       real sum;
      ┌──────────────┐
 ───→ │let i = 1;    │
      │let sum = 0;  │
      └──────────────┘
       Loop:  if i > n then go to Exit;
              sum = sum + A [i];
              i = i + 1;
              go to Loop;
       Exit:  return (sum);
   end;
```

Figure 3.9 Search Answer

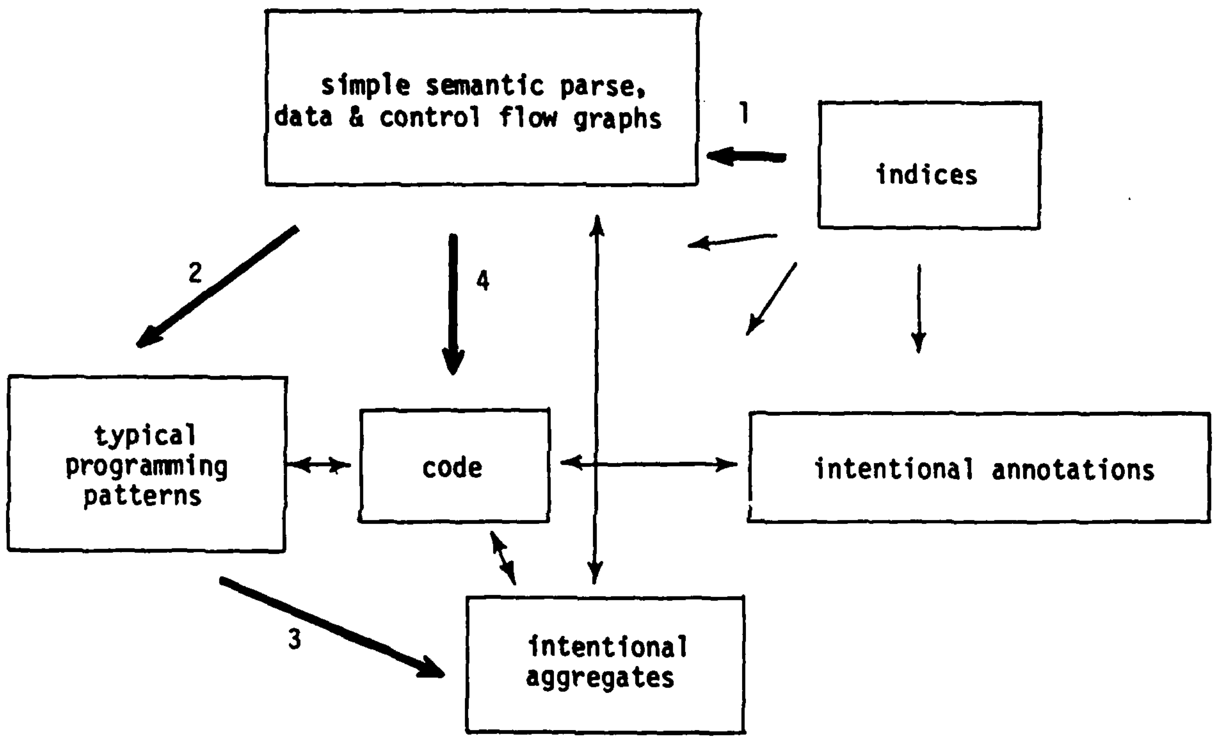


Figure 3.10 Solution Path

Once the search mechanism has applied all of the constraints applicable at this level, it follows links from the remaining segmented parse structures to the next higher level of PRL data which might be involved. Here, these are the typical programming patterns which involve loops that contain initializations. (Note that many TPPs might be retrieved. A given loop may implement a number of functions simultaneously, or be one element of a hierarchy of programming functions, all of which would be retrieved.) One of the elements returned would be an instance of a summation TPP. This template explicitly identifies the portions of the segmented parse that fill the required and optional roles of the summation TPP, shown in figure 3.6. At this stage of the problem solving process, all loops which compute sums have been identified.

The next step is to search among the summation loops for the one which computes the sum of the test scores. This is accomplished, once again, by following upwards pointers to the intentional aggregates which involve summation TPPs. At this level in the PRL hierarchy, there are annotations which map program structures onto the application domain concepts they represent (see figure 3.7). In this case, there is an aggregate for a statistics program that analyzes intelligence test scores. In it there is a reference to a variable which contains the "sum of the test scores". By textually matching this description against the search request, the system completes the process of identifying the structures relevant to the problem at hand (see the solution tree in figure 3.8).

The remainder of the solution process is simple. By examining the segmented parse in figure 3.5, the PRL can trace the data flow links from what is now recognized as the sum calculation in the augmentation body of the loop to its initialization segment (and from the index generator to its initialization as well). The initialization segments contain pointers into the code, which makes it possible to print the answer shown in figure 3.9.

The four-step search path discussed above is summarized in figure 3.10. It is one route among many that the PRL search mechanism might take to answer the original request, but it illustrates the type of activity involved. A number of knowledge sources are consulted, and their connectivity is employed. The search actively uses both detailed and descriptive (goal-oriented) information about program structures, which the PRL supplies. Lastly, this example shows that the PRL's knowledge is represented in a general way, which will allow it to support a wide variety of referencing requests.

4. The Program Reference Language Implementation

In order to demonstrate the feasibility of the IPE, we are implementing a search facility which operates on a program knowledge base like the one described in the previous chapter (see figure 3.3). This system, called the PRLI, is essentially a subset of the PRL; it allows the user to identify a region of code based on a complex syntactic and semantic description, but it stops short of automatically implementing such requests, as the full program reference language would do. In this demonstration, the user directly accesses the EPM's knowledge sources, and applies a detailed understanding of their structure to locate the specific portion of the program he wishes to find. The PRLI provides a simple query language which aids this process.

We chose to implement this search facility because we feel that its capabilities are logically central to the IPE. In a purely mechanical sense this is true, because the ability to isolate portions of programs must exist before editing knowledge can be applied to transform them. However, on a more theoretical level, the creation of the IPE hinges on the research issues involved in the creation, maintenance, and use of multiple sources of knowledge. For this reason, we see the task of implementing the PRLI as the simplest way to test our understanding of these problems; to build the system, we have to develop a collection of representations that capture program features, and we have to coordinate the information they contain in order to satisfy search requests.

The demonstration system has three subparts (please see figure 4.1): a knowledge base containing the program representations described in the previous chapter, a code region abstraction which allows the user to represent the arbitrary sections of programs obtained during the search process, and a user interface which provides a method for manipulating sets of data retrieved from the knowledge bases. This implementation has four knowledge sources, corresponding to the textual, syntactic, segmented parse, and typical programming pattern viewpoints. We have omitted the intentional aggregate and intentional annotation data, as well as informal textual comments, because they would require more detail, vs. more innovation to include.

Each of these knowledge sources is implemented as a data abstraction with its own set of operations for examining, and searching among its contents. For example, the text abstraction has the standard string search capabilities available in editors. The user can move forward to the first occurrence of the letter "x", or backwards to the second instance of the string "(integer: y);"). Syntax tree operations correspond to the types of search procedures which might access an ADA DIANA tree. They are able to locate the "begin" block within the current frame, or find the declarations of procedure bodies. There are similar operations on the other data types. In addition, each knowledge source has an index structure which allows certain subclasses of its objects to be generated trivially. (As with most data base systems, the choice of indices is at the discretion of the system builder. In general, we have provided the ones which looked the most useful.) So, for example, it is possible to locate all typical programming patterns which are of type "summation", and all segmented parse frames which represent generators. These search and indexing mechanisms are actually a bit more complicated, but they will be described in more detail when each of

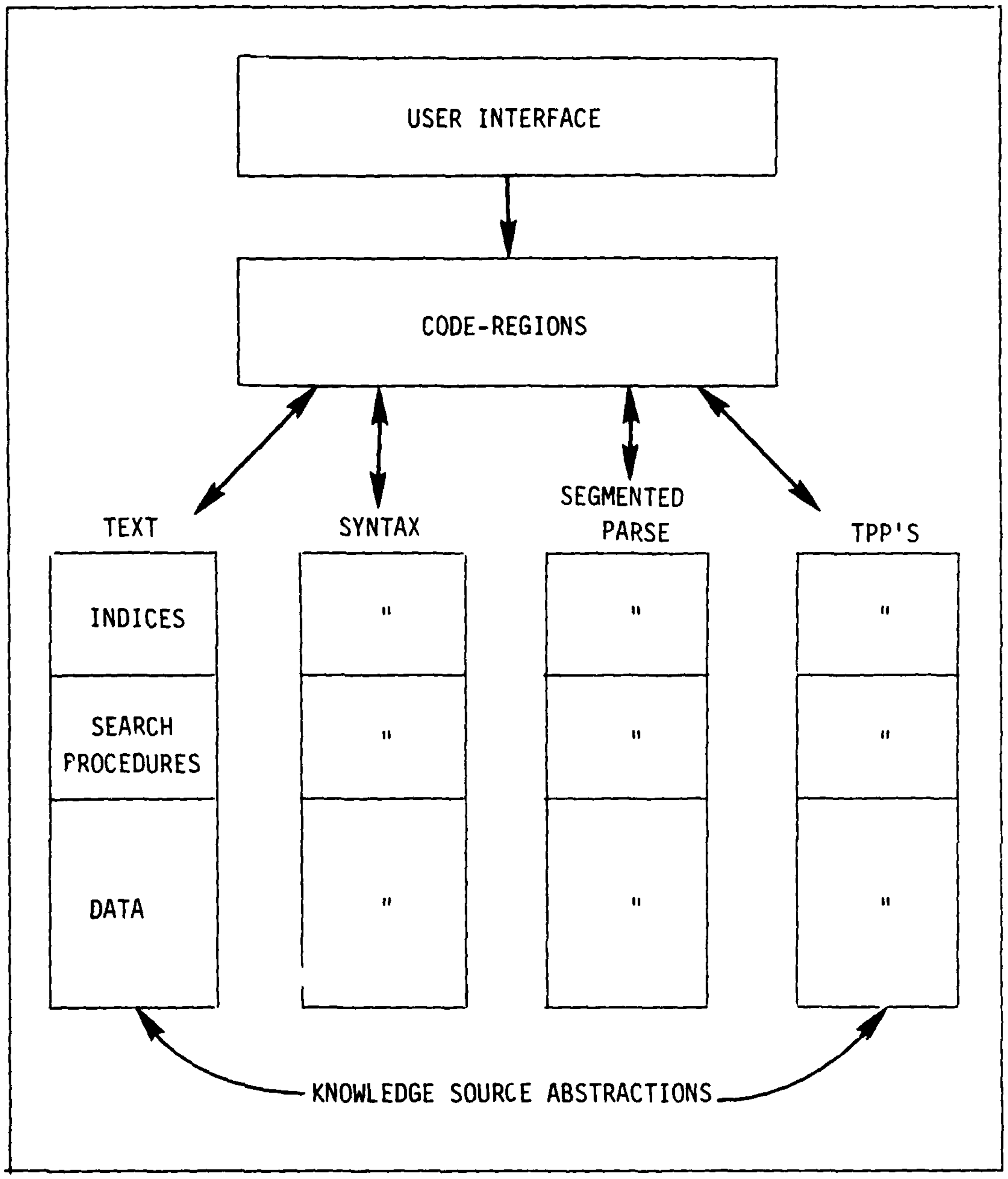


Figure 4.1 PRL Implementation (PRLI)

the knowledge sources is discussed in turn.

4.1. Code painting: the search approach

There are two general methods of identifying regions of code which satisfy the user's specifications. In the first approach, which we call sequential filtering, the user makes a gross stab at selecting his code region by generating all the elements which satisfy a fairly gentle condition. For example, if (as in the scenario which follows) he is interested in the loop which computes the sum of the test scores, he might start by locating the set of all loops in the program (which might be trivially available from a prestored index) and then sequentially restrict that set by applying more and more conditions. In the second approach, the user retrieves some collections of items, possibly from several different knowledge sources, and then intersects them together to find the elements which satisfy all of the conditions he wants to impose. In the kind of interaction we are trying to foster, the programmer should be free to consult whatever type of information is most appropriate, and have those actions cumulatively identify the regions of code he desires.

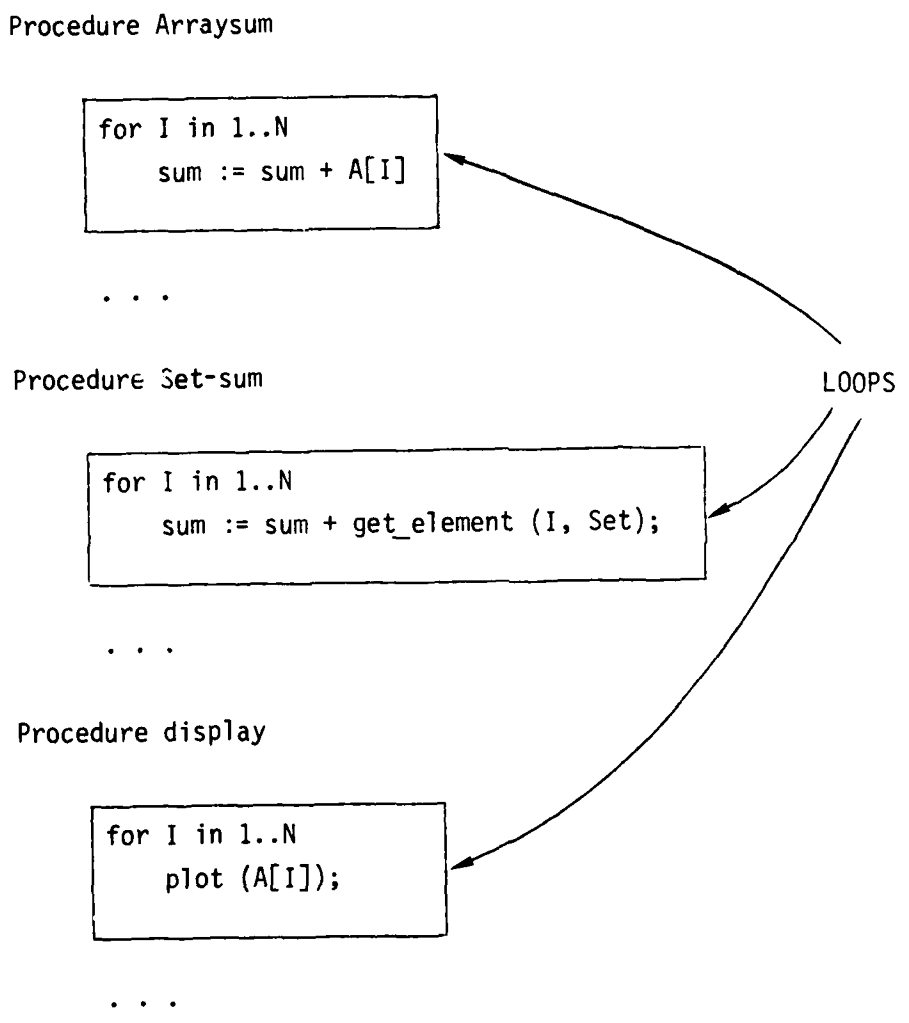
From a computational point of view, the main problem involved in either one of these approaches is to define a mechanism which is able to compare information obtained from the different sources of knowledge. To do so requires the ability to draw correspondences between the elements of different types, either directly, by linking the representations together, or indirectly, by defining a separate language which all of the different abstractions can "speak".

The PRLI accomplishes this interaction by viewing each element of the different knowledge bases as a specification for a region of program text (meaning a portion of the program listing); it combines them by overlaying the corresponding sections of code. This paradigm is called code-painting, and is described further below.

Figures 4.2 through 4.5 give an example of code painting applied to the search request shown earlier in section 3.3. In that scenario, the user was attempting to locate the "initializations of the loop that computes the sum of the test scores" using a sequential filtering paradigm. Here he might identify the set of loops in the program by consulting the syntactic knowledge base (figure 4.2), the set of summation templates by searching the TPP knowledge base (figure 4.3), and the places where the data from the variable "test-scores" is used by accessing the segmented parse representation for the code (figure 4.4). When these are overlayed, only the region corresponding to the sum of the test scores is marked in all three ways. (Please see figure 4.5.). Figuratively speaking, the PRLI colors one section of the program red, and paints the second code region yellow, and any region which comes up orange has all of the properties that were desired.

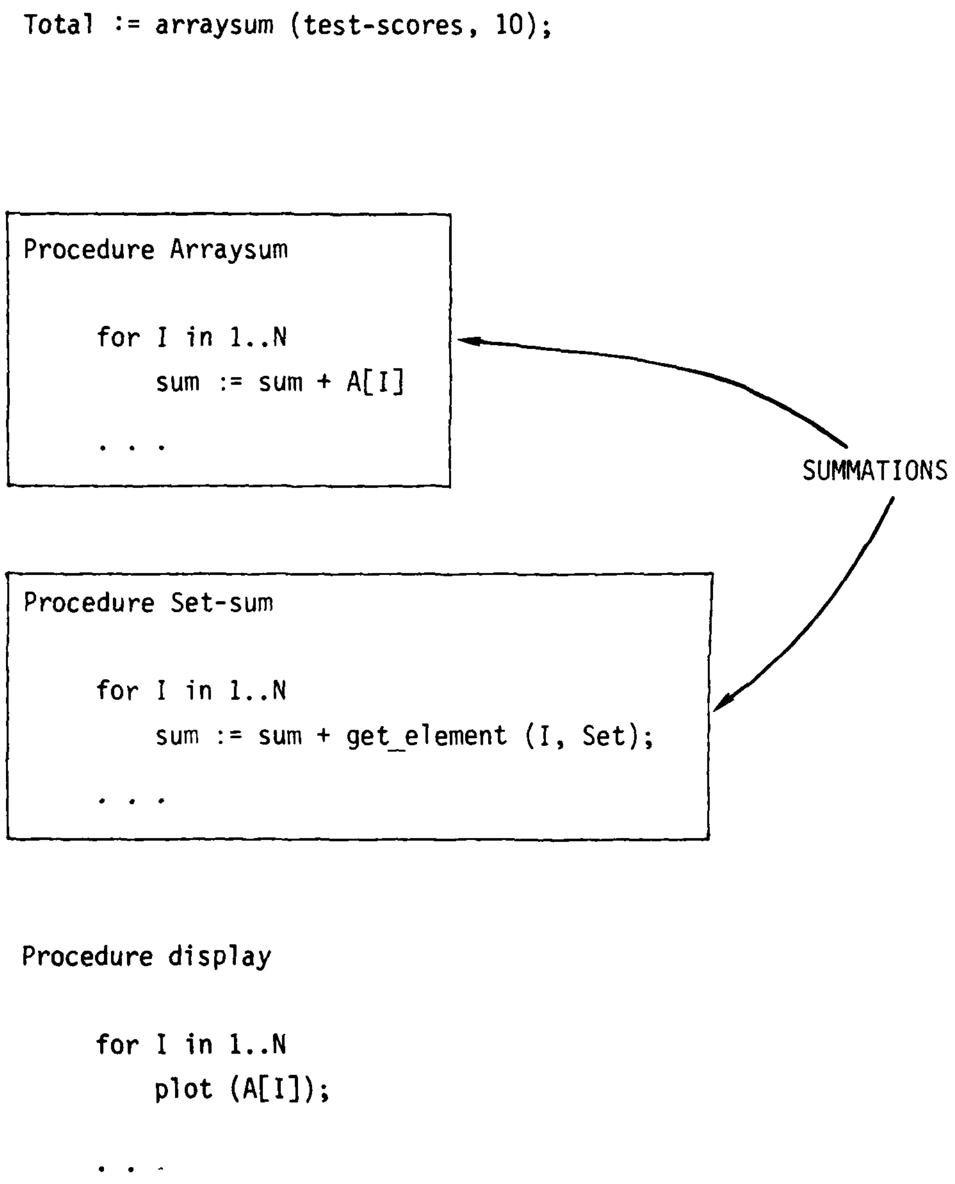
Code painting is implemented by the code-region abstraction shown in figure 4.1. In the PRLI, the user only sees code regions, and the demonstration system automatically accesses the appropriate knowledge source when one of its operations is applied. In order to accomplish this change in perspective, each knowledge base element contains a pointer that explicitly identifies the code region it corresponds to. Similarly, the code regions contain backpointers to the knowledge source

```
Total := arraysum (test-scores, 10);
```



```
Procedure Arraysum

    for I in 1..N
        sum := sum + A[I]

    . . .

Procedure Set-sum

    for I in 1..N
        sum := sum + get_element (I, Set);

    . . .

Procedure display

    for I in 1..N
        plot (A[I]);

    . . .
```

Figure 4.2 Occurrences of Loops



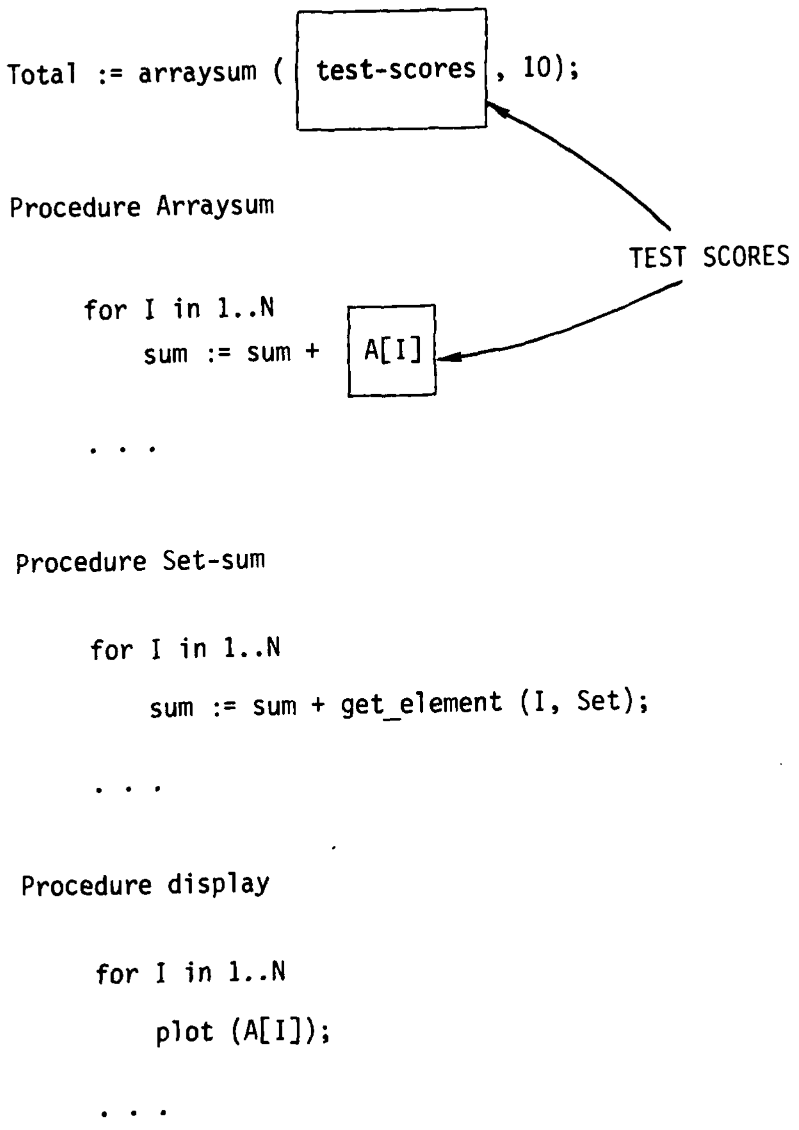
```
Total := arraysum (test-scores, 10);
```

```
Procedure Arraysum

     for I in 1..N
         sum := sum + A[I]
     . . .
```

```
Procedure Set-sum

     for I in 1..N
         sum := sum + get_element (I, Set);
     . . .
```

```
Procedure display

     for I in 1..N
         plot (A[I]);

     . . .
```

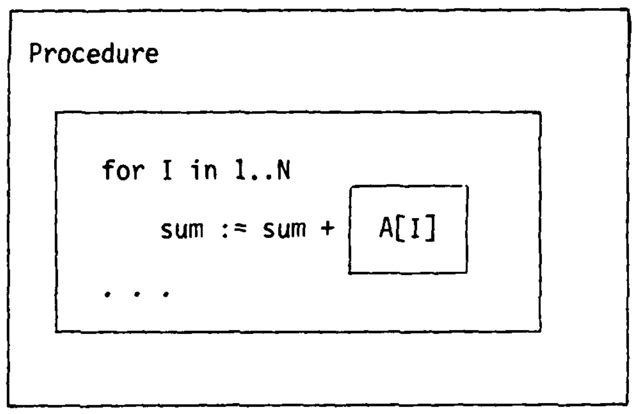
Figure 4.3 Occurrences of Summations



```
Total := arraysum ( test-scores , 10);

Procedure Arraysum

     for I in 1..N
         sum := sum + A[I]

     . . .


Procedure Set-sum

     for I in 1..N

         sum := sum + get_element (I, Set);

     . . .

Procedure display

     for I in 1..N

         plot (A[I]);

     . . .
```

Figure 4.4 Occurrences of Test Scores

```
Total := arraysum (test-scores, 10);
```



```
Procedure

    for I in 1..N
        sum := sum + A[I]
    . . .
```

The code region which computes the sum of *the test scores.*

```
Procedure Set-sum

    for I in 1..N
        sum := sum + get_element (I, set);

    . . .


Procedure display

    for I in 1..N
        plot (A[I]);

    . . .
```

Figure 4.5 The Intersected Code Region

entities over which they are defined.

Code painting is, however, a deliberately coarse affair. It was designed as a mechanism for exploiting imprecise and partial information, which is what we expect to be available in all reasonable implementations of the PRL and IPE. This is true because the correspondence between knowledge source entities and code regions is not always well defined, and in the context of the program reference language (which is meant to use currently available technology), even if it is conceptually available, it cannot always be automatically derived. A case in point is the typical program pattern knowledge base. In the absence of an automatic recognition facility, there is no way other than asking the user to identify programming cliches. The system can maintain a library of TPPs, and make them easy to instantiate, but in any situation where the user is called on for unverifiable information, it will almost by definition be approximate.

It is interesting to note that our design philosophy even encourages users to provide low quality information if it is all that is available or if it was simply convenient. We take this tack, because we believe that people will not use a system that contains a restrictive interface, which demands the user to provide a great quantity of detail. On the converse side, the performance of the PRLI should increase with the presence of better documentation. In this way, we hope to motivate programmers to supply more accurate information.

The PRLI is particularly suitable to a coarse approach because its task environment is both error tolerant and forgiving. It is error tolerant in the sense that if a code region is returned to the user as the answer to his request, it is acceptable if it is wrong 10% of the time. The search process is forgiving in that answers can be almost correct. E.g., if the user is looking for the section of his program which iterates over the items of a particular list, and the PRLI responds with the code region for the enclosing block, the answer is sufficient. (The desired code will physically be present on the screen.) For these reasons, we have deliberately conceived of the code painting paradigm as a broad spectrum approach, which applies a thick brush in the process of answering user queries.

The following sections describe the individual components of the PRLI, shown in figure 4.1. Each of the knowledge sources is discussed in turn, and specific examples of their data structures are provided. However, at the current time, implementation of the PRLI is not quite complete. As a consequence, the scenarios provided later in the chapter are hypothetical; they have not been run.

4.2. The knowledge sources

The implementations of each of the knowledge sources share some features in common. In particular, the syntax, segmented parse, and TPP abstractions are all composed of frames which have various attribute-value pairs. These frames contain code-regions, but they do not point to one another as in the overview provided in chapter 3. In addition the PRLI contains a generic search mechanism which can filter elements derived from the different knowledge bases; it allows frames to be selected by attribute-value pairs. For example, given a set of syntax frames, the filtering mechanism can pull out the ones of type "procedure". Given a segmented parse node, it can tell if it corresponds to a terminator, or if it inputs a specified object. The composition of each of the data type frames is discussed below. What is important to note here is that the knowledge sources do not contains search procedures so much as they provide generators which produce subsets of their nodes. These subsets can then be pruned using the filtering mechanism already described.

The data which is contained in the different knowledge sources is also subject to a strong constraint; namely, that it could either be generated automatically using tools in today's technology, or it could be reasonably supplied by the user of the PRLI. Because of this imposed limitation, the system's knowledge is both approximate and incomplete. The example frames taken from the knowledge source exhibit this effect; it is up to the code-painting paradigm to use the information in the best way it can. All of the examples below relate to the search scenario shown in chapter 3.

4.2.1. Program text

The text abstraction in the PRLI represents programs as a conventional editor would do. It holds the text in a doubly linked list of lines, each of which can contain up to some fixed number of characters. The text abstraction has its own search operations defined on it, which scan forward and backward for particular character/delimiter sequences. There is a cursor which holds the current location. So, for example, the user can search for the first occurrence of the string "xyz;", or backwards for the second instance of "a b c".

The fundamental unit of text which can be mapped into code regions is the token, defined as an alphanumeric sequence separated by delimiters. The user can ask for the token surrounding the cursoı ınd convert it into a code region, although that mapping is somewhat indirect. It turns out that for reasons of efficiency, text lines are associated with code regions, and the regions for tokens have to be located by scanning within those.

4.2.2. The syntax tree

The syntax abstraction is represented as a tree of frames, where each frame corresponds to a clause in the syntactic parse of the program. So, for example, there are nodes for "procedures", "formal parameters", "labels", "while statements", etc., while the terminals of the tree correspond to tokens. An example frame is shown in figure 4.6.

The syntax tree has a generator defined on it which is able to produce all nodes in the linked structure below a given frame. In addition, it has an associated index which can produce the set of "procedure" nodes, and the set of "iteration statement" nodes (meaning the while-loops and for-loops in the program).

The syntax tree forms the basis for the implementation of code-regions. This was done for two reasons; first, as a hierarchical structure, the syntax abstraction provides a convenient method for identifying subset and superset relationships which is needed in order to intersect code regions. Second, the syntax tree is essentially a static structure that is updated only when program modules are changed. For

```
TYPE: "For loop"

Parent Frame:  [begin block 2]

Subframes:  [token "for"], [iteration variable 1],
            [iteration range 1], [loop body 1]

Segmented parse pointer:  [loop1]

TPP pointer:  none

Code region:  <for I in 1..N...>
```

[ ] = Syntax tree, or segmented parse frames

Figure 4.6 The Syntax Tree Frame for For-Loop #1

this reason, the individual syntax nodes contain the backpointers that associate code regions to elements of the other knowledge source abstractions.

4.2.3. The segmented parse

The segmented parse representation is a somewhat simplified version of the diagrams presented in figures 3.5 and 3.6. It is represented as a frame with a number of attributes that correspond to the type, inputs, outputs, subsegments, code region, and operation which the given segmented parse node performs. The type gives the element of vocabulary which the segmented parse node fulfills. So, for example, the type might be a generator, a filter, a loop, a predicate or an expression. The operation might be "+" or ">", or rather, the code regions they correspond to. There is no operation for a non-terminal node. The subsegment field contains the segments which the given node lexically encloses. For example, a loop always contains a generator and a terminator.

The input and output fields are filled by somewhat more complex entities called ports, which trace and order the data flow between segments. Ports stand for the input arguments and return values of functions. Each port knows the segment it is on, the port which has data flow to it, and the ports to which its data flow goes. Where applicable, ports contain the variable names associated with their data flow in the program.

The segmented parse abstraction has several generators defined on it. It can produce its frames by lexical enclosure (subsegment relationships), or by following the data flow from a given node. We have temporarily ignored control flow as a method of generating or selecting nodes. The knowledge source also has three indexed properties: ports can be generated given a program variable name, procedure names are associated with their corresponding segments, and serve as indices to the segments which represent calls on them as well.

An example of a segmented parse frame is shown in figure 4.7.

4.2.4. TPPs

Typical programming patterns technically represent mappings from named roles to segmented parse frames. Ideally they would be defined in that fashion, in order to gain the explicitness provided in that abstraction which is useful for an in-depth understanding of code. However, in the context of the PRLI, the user is responsible for instantiating TPPs, and we felt that it was inappropriate to expect that quantity of detail. In our model, TPPs are linked directly to code regions. They contain named roles for the actions of particular programming patterns which are associated with code-regions as well.

TPPs are represented as frames, again with attribute-value pairs. An example is provided in figure 4.8. Here, there is not much information provided, other than that the indicated code region can be referred to as a summation, and that the data containing the sum, and the operation computing the "+" are identified in the code.

In a further implementation, there will be two types of TPP frames; templates making up a library for typical programming actions, and

```
TYPE:  "LOOP"

OPNAME:  NAME

INPUTS:  PORT1:                                  PORT2:
              OWNER:  [LOOP1]                        OWNER:  [LOOP1]
              DFLOW IN:  port1 of [EXPRESSION1]      DFLOW IN:  port2 of [EXPRESSION1]
              DFLOW OUT:  port1 of [TERMINATOR1]     DFLOW OUT:  port1 of [ACCUMULATOR1]
              NAME:  "N"                             NAME:  "A"


OUTPUTS:  PORT3:
              OWNER:  [LOOP1]
              DFLOW IN:  port3 of [ACCUMULATOR1]
              DFLOW OUT:  port3 of [EXPRESSION1]
              NAME:  "SUM"


SUBSEGMENTS:  [GENERATOR1], [TERMINATOR1], [ACCUMULATOR1]

CODE-REGION: <for I in 1..N loop ... end loop>

PARENT SEGMENT:  [EXPRESSION1]
```

< > = Code region for the enclosed code

[ ]   Segmented parse frames

Figure 4.7 The Segmented Parse Frame for Loop #1

```
TYPE:  "SUMMATION"

CODE REGION: <Procedure ARRAYSUM...>

LOOP EXPRESSION:

GENERATOR:

TERMINATOR:

ACCUMULATOR:  <SUM := SUM + A(I)>

ARRAY:  <A(I)>

RESULTS:  <SUM >
```

< > = Code region for the enclosed code
Empty slots have not been instantiated by the programmer.

Figure 4.8 A Summation TPP

instantiated TPPs representing specific portions of the user's code. The library templates will also have constraint information attached. For example, the constraints might force the user to fill in certain slots in order to accept a template, such as the identity of the data flow which holds the sum of the summation. These TPPs will also carry consistency constraints, which might say that the body of the generator in a summation must have data flow to the body of the accumulator, if the TPP is to be valid. This information could be used to prompt the user, or automatically make deductions in addition to identifying inconsistencies.

The TPP structures have two indices; the sum total of all TPPs defined over the program can be produced in some order, and the TPPs of a particular type can be found. TPPs are not hierarchical structures, and hence they cannot be generated by following links within specific frames.

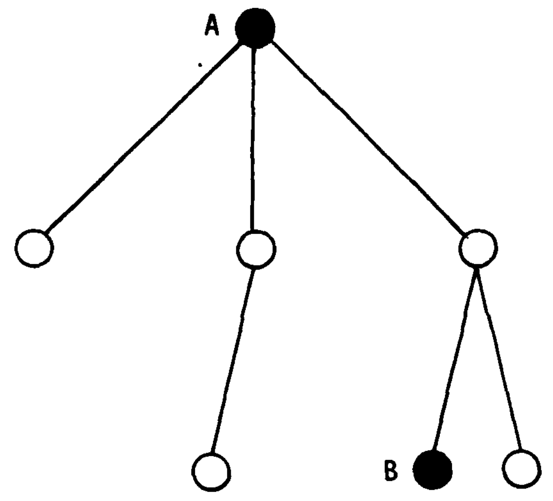
4.3. Code regions

Code regions provide the mechanism used to combine information obtained from the knowledge sources in the PRLI. It accomplishes this by the paradigm of code painting (described above) which relies on the ability to intersect regions of code. In code painting, the system overlays the sections of the program which given knowledge base elements correspond to. This has the effect of "ANDing" together the search requests which were used to obtain those knowledge base elements to begin with.

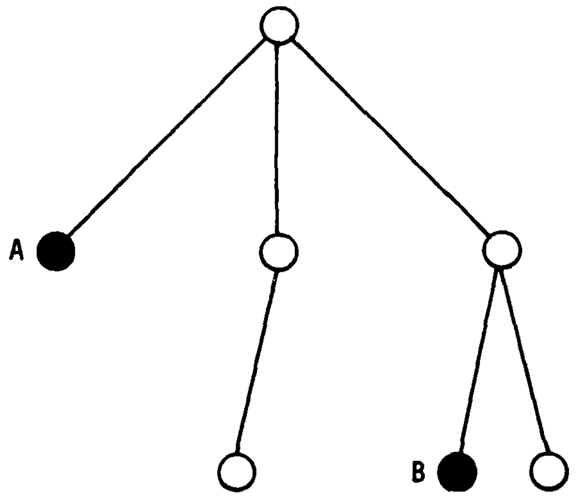
A code region is defined to be a set of syntax tree nodes. It corresponds to a contiguous, or non-contiguous region of program text. For example, a code region might correspond to a single syntax tree node, such as a terminal frame for the token "sum", or it might designate a complex region built out of a "procedure" node together with a separate "iteration" construct. In both cases, the effect is exactly equivalent to underlining portions of text. (This is always true since the syntax tree contains all tokens in the program.) We use syntax tree nodes because they are more efficient and invariant for the purpose of computing intersections, than, say, program text.

The mechanism for intersecting code regions makes use of the hierarchical nature of syntax trees. The PRLI accomplishes it in the following way (please see figure 4.9): If two code regions are primitive, meaning that they both correspond to single syntax tree nodes, then it must be the case that either they are completely independent, they are the same node, or one of them is syntactically a part of the other. Their intersection is either null, or the node which is totally contained (the subset node). Therefor, the intersection paradigm proceeds by testing to see if node A is a subset of node B, or if node B is a subset of node A. This subset test is accomplished by scanning up the tree from one node to see if the other is reached before the root of the tree. If A is found above B, then B is a subset of A and B is the intersection of A and B.

The approach extends to the problem of intersecting code regions composed of sets of syntax tree nodes. However, one additional constraint has to be imposed; namely, the nodes comprising any single code



B is a Subset of A



A and B are Independent

Figure 4.9 Two Single-Node Intersection Cases

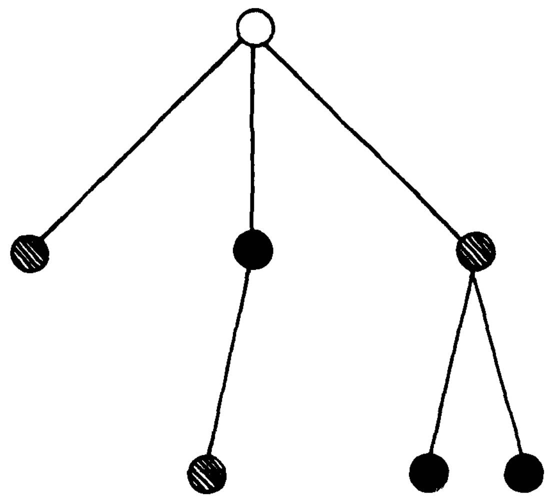
region must be disjoint. Under this restriction the mechanism described above can be used to compute the intersection of two complex code regions, X and Y. It is composed of all elements of X which are subsets of Y, and all elements of Y which are subsets of X. See figure 4.10. (There are some computational short-cuts available, but we will not go into those here.)

Code regions have additional operations for taking the union of two regions, and for aggregating small sections into larger contiguous ones if they are defined. These operations are also implemented in terms of the intersection, or subset computations described above.

In addition, code regions contain pointers back into the knowledge source entities which reference them. These connections are needed in order to complete the correspondence between the different program viewpoints. However, they also have somewhat of an imprecise flavor; the correspondence is not always well defined, and again, the information may be imprecise. For example, in order to find the segmented parse structures over a given code region, all of the syntax tree nodes in the region have to be examined in turn. If one of them does not contain a back-pointer, then the PRLI might use a heuristic approach. Here, it could apply the back-pointer contained in the node's parent, or in one of its immediate successors. It is not clear that one of them is a better approach.

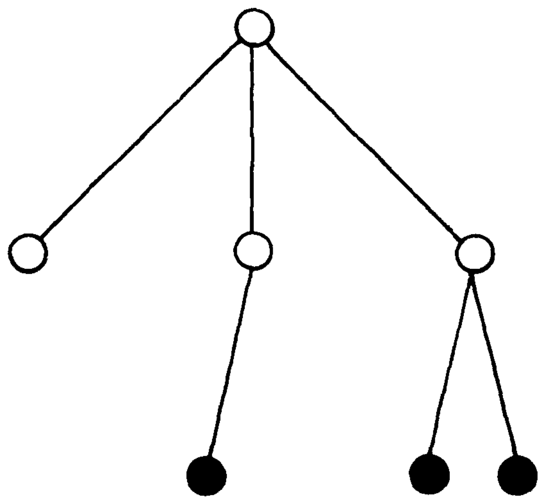
4.4. User interface

The user interface of the PRLI only provides the capabilities



▨ = Code Region A

● = Code Region B



Intersection of A & B

Figure 4.10 Intersecting Two Code Regions

necessary to manipulate the data obtained from the different knowledge bases. In particular, it defines a standard set abstraction which is used to communicate the results of the different queries from the search procedures to the user.

This set abstraction has the operations of member, union, intersect, subset, add, remove, first, rest, and length. These should be self-explanatory. In addition, the abstraction allows a series of operations to be applied in turn to the elements of a set. The operations can include the attribute-value filters mentioned earlier. For example, if the user is interested in finding the loops which are summations that contain initializations, he might type the expression

set-apply (*all-loops type=summation subseg type=initialization*)

This invocation would cause the following actions: first, the loops in the set "all-loops" would be pruned to the ones which define summation templates. Second, the PRLI would find the subsegments of that set (this would cause an internal conversion to the segmented parse viewpoint), and finally the segmented parse nodes just obtained would be pruned to the ones which were initializations. The code regions corresponding to the final set of initializations would be returned.

The user interface to the PRLI always presents the programmer with code regions as the resolution of his search requests. The various conversions between the knowledge based viewpoints happen automatically, and are keyed by the type of the operation requested. For example, in the search expression above, the "type=summation" filter causes the system to convert the code regions it internally uses into the TPP templates which are defined over them. Similarly, the "type=initialization" filter causes a switch to the segmented parse viewpoint.

The programmer, however, still has to be conscious of the structure of the PRLI's knowledge base. He needs to know what types of viewpoints exist, and also the the attribute-value combinations available in the different abstraction frames. He is shielded only from the need to explicitly convert between the representations in order to use their properties.

In future versions of the PRL, there will be a more uniform search language for accessing the program knowledge which the system contains. The interface will support all of the terminology we have defined, but it will also contain the knowledge necessary to implement search requests automatically, without exposing the structure of the knowledge base to the programmer. This system would be a complete program reference language, and it would rely on the type of the data base retrieval mechanisms described here.

4.5. A Scenario using the PRLI

This scenario concentrates on the set intersection method for isolating a program portion. Here, the user characterizes a piece of code by several independent methods, and essentially "ANDs" them together to locate the program fragment which answers all the descriptions. Once again, this scenario is hypothetical, although most of the functionality it is based on has been implemented in code. The query, and final answer are the same ones described in the previous section.

User queries will be shown in lower case, while system output will be displayed in capital letters. Our commentary is enclosed in brackets.

```
> sums:= index ("summation", Tpps)
-> {TPP-CR1,TPP-CR5}
```

[the user asks for all TPPs which are of type "summation". This is an indexed property of the TPP data base, meaning that all TPPs can be located by their type, without having to search through the entire knowledge base. The answer is a set of code regions corresponding to the TPP objects called "sums".]

```
> loops:= index ("loops", Stree)
-> {CR-STREE1,CR-STREE2, ...}
```

[Similarly, the user examines the syntax tree ("STree") for all elements of type "loop". The result is a set of code regions for syntax tree nodes, one for each For loop and While loop, etc., that are in the program. "..."s represents ellipsis, meaning that the PRL returned a set of more than two elements, all of which are not being printed.]

```
> tscores:= index ("test-scores", Segp)
-> {CR-SEGP3, CR-SEGP5, ...}
```

[The segmented parse ("Segp") data base indexes its frames by the data they input or output. However, this is a data flow indexing as opposed to a variable indexing. This means that the user retrieves all places where the data from the variable "test-scores" is used, in addition to those frames where it is referred to by name. For example, subroutines which were called with the test-scores data would also be retrieved. Note that the user could have been more specific by requesting only those segmented parse frames which output the test score data. This may not have been an indexed property of the segp data base however, requiring the user to either filter through the segp set returned above, or filter the entire segmented parse data base for frames with the proper attributes.]

```
> answer:= cr-intersect (sums, loops, tscores)
-> {CR1}
```

[Here, the user computes the intersection of the three sets. However, the operation is somewhat different than normal set intersection; it is based on properties of the objects rather that their actual presence as literals in each of the sets. In our implementation, the set of sums, loops, and tscores can be viewed as gateways into the code. They stand for the particular pieces of the program which can be described as a part of a summation computation, or a portion of a For loop, or as a piece of code which uses the data from the variable "test-scores". Therefore, the intersection operation has the effect of intersecting the descriptions, or of locating the items which can be described in all three ways. In English this corresponds to identifying the loop which computes the sum of the test scores.

Internally, the following actions are taking place. First, the Tpp, Stree, and Segp sets are converted into their corresponding Code Regions (CRs). This operation happens automatically (it is keyed on the fact that cr-intersection only takes CR inputs), and is accomplished via the pointers maintained in the frames of each knowledge source that explicitly indicate the code regions they represent. (The origin of these structures is discussed in a previous section.) Following this, the code regions are intersected using an algorithm which effectively paints sections of the program text. The segments which are painted with all three colors (for the three input code regions) are the ones which survive the intersection. In the case above, there is only one code region remaining.]

```
> select ("initialization", answer)
-> {SEGP22}
```

[The user is interested in the initialization of the loop which computes the sum of the test scores, and specifically asks for that frame to be returned. The concept of an "initialization" is a vocabulary element provided by the segmented parse abstraction, and so the system automatically converts the answer obtained in previous query to the Segp form. This type conversion is made possible by virtue of the back-pointers contained within code regions, which were installed when the code regions were created.

Note that it is in general possible to go backwards from code regions to knowledge sources due to the fact that code regions are implemented as collections of syntax tree nodes. Since the syntax tree is a more static structure, it can be set up with inverse pointers when the knowledge sources are initially created, as opposed to the process of merging the dynamically created code regions in with the remainder of the knowledge base. This topic is discussed more thoroughly elsewhere.]

```
> (print *)
->  Procedure Arraysum (A:Array, L:integer)
        /*L is the number of array elements to sum*/

  >>>   L:= 0
        ...
```

[The user prints out the code corresponding to the initialization, pointed to by ">>>". This again involves implicit type conversions from the segmented parse to code regions, to the text representation for the program.]

5. References

1. Dean, Jeffrey, McCune, Brian P., "Advanced ADA Tools for Software Maintenance", AI&DS Technical Report 3006-1, October 1982.

2. Rich, Charles, "Inspection Methods in Programming", AI-TR-604, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., 1981.

3. Shapiro, Daniel, "Sniffer: A System that Under ... ds Bugs", AIM-638, Artificial Intelligence Laboratory, ... chusetts Institute of Technology, Cambridge, Mass., 1981.

4. Waters, Richard C., "Automatic Analysis of the Logical Structure of Programs", AI-TR-492, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., 1978.

DISTRIBUTION LIST

Procuring Contracting Officer
Office of Naval Research, Code 614A
800 N. Quincy St.
Arlington, VA 22217
Attn: David K. Beck
(1 copy)

Scientific Officer
Communication & Computer Tech. Proj. Mgr.
ATTN: Mr. Joel Trimble
Office of Naval Research
800 N. Quincy St.
Arlington, VA 22217
(2 copies)

Dr. Robert B. Grafton
Office of Naval Research, Code 433
800 N. Quincy St.
Arlington, VA 22217
(2 copies)

Defense Technical Information Center
Cameron Station, Bldg. 5
Alexandria, VA 22314
(12 copies)

Naval Research Laboratory
Code 2627
Washington, D.C. 20375
(6 copies)

Office of Naval Research
Western Regional Office
1030 E. Green Street
Pasadena, CA 91106
(1 copy)

MED

82